Bootstrapping
Research
in Computer Science Education

The Experiment Kit

Experiment Kit: TOC

1. Research question (justification of method)
2. Protocol
    a. Human Subjects materials
    b. Background questionnaire
    c. Discriminator question
    d. Specification of setup
    e. Experimenters' script (including guidance on notes/diagramming)
    f. Participant "introduction" instructions
    g. Stimuli set
    h. Stimuli key
3. Analysis protocol
4. Background
    - Details of Pilot Study
    - Literature that contributes to the question
    - Literature that contributes to the methodology
    - Background literature

1. Question Formulation

General research question

This kit addresses programming understanding and conceptual foundations of programming skills in first- and second-year undergraduate students, those we term 'first competency' programmers (using McCracken et al.'s formulation ITiCSE, 2001). The study aims to elicit students' understanding both of program construction (i.e., process abstractions) and of programming concepts and constructs (i.e., conceptual structure of the domain).

Study's focal question

What sense are students making of programming concepts? Do they have meaning for individual concepts (e.g., tree), do they meaning for affiliated concepts (e.g., tree and list are data structures), and do they have meaning for relationships among groups of affiliated concepts (e.g., data structure vs. control structure), i.e., are they making sense, are they forming abstractions, and are they deriving conceptual structures?

Links to relevant theory

Why care about abstraction formation?

- One consistent result in the expert-novice literature is that experts organise or 'chunk' information differently from novices, that they form abstractions based on deep (semantic, functional) characteristics rather than on surface (syntactic) characteristics. (Summarised in Kaplan et al., 1986; Allwood, 1986).
- 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 "deep learning" characterised by abstracted meta-knowledge. (For an overview see, e.g., Eysenck and Keane, 1995).
- In a related study, one which uses a similar approach, the relationships exposed by a categorisation task were taken to reflect relationships in the subjects' internal representations and hence to lead to a model of their internal representations (Adelson, 1981).

Study's approach

This study uses a categorisation task – a repeated, single-criterion, card sort – to elicit students' articulable knowledge of the programming concepts presented on the cards as stimuli. It uses laddering as a secondary method, to elicit elaboration and clarification of the categories identified in the card sort.

Assumptions:

- By putting the card into a 'meaningful' group (i.e., not 'not applicable' or 'don't know'), subjects are demonstrating that the concept represented by the card has some meaning for them (i.e., they are adding something to what they are given).
- The category-criterion relationship reveals something about the subjects' construction of the domain. In this context, assigning cards to *any* category (i.e., to any named group of cards) says something about what subjects consider the criterion to represent and how they interpret it.
- By asking subjects to do multiple, single-criterion sorts, we're asking them to articulate elements of their conceptual structure of the domain and/or their cognitive structure of the activity of program construction.

Justification of approach:

- *Categorisation is appropriate for elicitation*
  This study uses a categorisation task to elicit conceptual relationships between the constructs presented on the cards. Categorizations tasks (Rugg and McGeorge, 1997):
  - can provide insights into the higher-level construing used by the respondent,
  - are appropriate for a range of entities, including abstract or complex entities.

- *Task independence*
  The focus is on abstraction formation, and hence we investigate whether there is evidence of sense-making outside specific task contexts. Given prior studies which indicate that comprehension strategies are task dependent and that categorisation might be (e.g., Davies et al., 1995), this study takes an approach which is independent of any particular programming task.
- *No imposition of existing overview or categories*
  The general aim of this study is to attempt to elicit knowledge which is not constrained by a pre-defined scheme nor by syntax specific to any one programming language (i.e., it tries to avoid imposing our own overviews and to avoid superficial language-bias). Therefore, we use multiple, subject-defined, single-criterion card sorts, and do not introduce any categories or criteria.

Subsidiary queries:

- *At this level*, which is more evident: the structure of the domain (consolidated knowledge), or the structure of the teaching (procedural and declarative knowledge).
- Are there qualitative differences between:
  - high- and low-performers?
  - genders?
  - first programming languages?
  - instructional approaches?
- Given that we don't know where the subjects' categories come from (i.e., the study only provides concepts; it is the subjects who add categorisation information):
  - What do their category structures resemble (e.g., expert models, instructional models etc)?
  - Are there common category structures across subjects (based on category content, not on criterion names)?
  - Are there indications of misconceptions or specific areas of ignorance, e.g., through oddities of categorisation?
- Given that we collect teacher data as well as student data:
  - Do students form categories more similar to their own teachers' than to other teachers' categories?

From where we stand here, it seems likely that these are questions that the data is going to invite us to investigate, but of course individual areas of expertise or interest from among the Bootstrapping participants should be followed up.

2a 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' conceptions and constructs of programming. Each subject will be asked to undertake a card-sort exercise, to be followed by an interview. These will take *circa* one hour each. During the sessions, data arising from the card sorts, and subsequent interview, 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 grade on programming courses – will be associated with the card-sort and interview 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 computer programming. This research is designed to investigate "first competency" programmers' conceptions and constructs of programming. You will be asked to undertake a card-sort exercise, to be followed by an interview. These will take *circa* one hour each. During the sessions, we will take notes from the card sorts, and ask that we may tape-record the subsequent interview.

Some personal data – your age, gender, institution and academic grade on programming courses – will be associated with the card-sort and interview 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.

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 understanding of computer programming concepts and program construction. I understand that data from the card sort and associated interview 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 any audio tapes of my conversations with the researcher will be erased no later than <insert date>.

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 <insert date>.

Signature
Date

2b Background questionnaire

For each institution:

- Characterise you 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).
- Describe the point of intervention (that is, "end of first year"; "half way through second year" etc.)
- Detail what programming courses the subjects have had up to that point. Note:
    - What language(s) they have had instruction on
    - What quantity of instruction they have had, stated as proportion of taught material within a full time student year. For example, at the point of intervention, subjects at the University of Kent had two courses of Java, which represents one quarter of the teaching for that year, and they have had and one course of Haskell which represents one eighth.

For each subject:

- Note their:
    - Age
    - Gender
    - Programming courses taken & grades achieved.
- Ask them to self-rate their programming experience on the form provided
- Assign each subject a unique identifier of the form: S01 (Student 01) or E01 (Expert 01), appended to your institution code (thus the first subject for the University of Poppleton would be HS01):

Programming Experience

On a scale of 1 (never used) to 5 (have used a lot) please rate your familiarity with the following programming languages:

	5	4	3	2	1
Java					
C++					
C					
Ada					
Scheme					
Pascal					
Visual Basic (VB)					
Other					

2c Discriminator Question

You should choose subjects from your undergraduate cohort at the point you believe them to be capable of undertaking at least one question within the McCracken task set (as used in *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students* ITiCSE 2001).

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.

2d 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 Consent Form
- The stimuli
- Participants' introduction instructions
- The stimuli key
- A working tape recorder
- Enough tapes & batteries
- Paper
- Pens

Sit so that you can see the subject (and they 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 at least 45 minutes for just a card sort. Allow an hour and a half if you are doing laddering at the same time. If you are conducting a separate laddering interview, then allow an hour for that.

2e Experimenters' script

Card Sort

The procedure is described in Rugg & McGeorge (1997)

Reiterations:

- We want single-criterion sorts – if you're in doubt that there is only one criterion, ask "Are all these categories of one criterion?" or ask a discriminator question, e.g.: "Are these categories all at the same level?" If not, ask subject to separate them into two sorts.
- Make sure to label the categories and criteria verbatim
- Don't suggest labels
- Be ready to take ancillary notes – descriptions, amplifications, etc.
- Note interventions such as triadic prompts, including which cards you used.
- label each page of your notes with subject identifier and sequence number (for recovering from dropping everything)
- Get them to read the numbers to you – but (if you can) also keep watch to see that the numbers they tell you are the ones on the card

Elaborations for this study:

- Recommended first intervention: encapsulation, scope, object
- Use triadic prompts for this study (note which you use)
- Make note of sorting behaviour: how they handle the cards, how they place them on the table (e.g., shufflers, dealers, spreaders, etc.)
- Try to log ephemeral sorts: if they appear to move the cards into groups on the table but don't articulate the categorisation, ask them to try to describe the sort, or ask them what sort of thoughts were in their minds when they produced that layout.

Laddering Prompts and notation

The procedure is described in the Rugg *Laddering* paper.

Reiterations:

- Laddering is a constrained technique. Conform to the notation and to the standard wording of the probes.
- Use their terms verbatim. Do not try to paraphrase or clarify. Add no interpretations.
- 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.
- Conduct the laddering as 'breadth-first'.
- Ensure that the subject has really 'bottomed out' before you stop.
- Verify the accuracy of what you're recording; say it back to the subject.
- Tape the session.

Opening prompts:

"What we're trying to find out is how people structure their knowledge".
"Let's unpack some of the concepts you used in the card sort".
"We'll work through systematically, so some of the questions may seem trivial."

Prompt for viewpoints:

"Can you tell me some viewpoints from which you might categorise this domain?"
or "You mentioned viewpoint A – are there other viewpoints you might use?"
Directing the main viewpoint: "I'd like you to answer from the viewpoint of someone constructing programs."

Confirmatory prompt: "Can I check that I've got this right now...?"

Notation for sub-class elicitation ("Can you tell me some sub-types of X?"):

_____	underlining identifies the class about which you're enquiring
;	delineator for each of the separate chunks within the response
[ ]	indicate a viewpoint

Notation for explanation elicitation ("How can you tell that something is an X?"):

_____	underlining identifies the class about which you're enquiring
;	delineator for each of the separate chunks within the response
[ ]	indicate a viewpoint
↓	explanation

Introduction

You will be given some cards to sort. Each card will have the name of a programming concept on it. We would like you to sort the cards into groups, using one criterion at a time. When you have finished sorting, we will ask you what the groups were that you sorted the cards into, and what the criterion was for that sort. Once this has been done, we would like you to sort the cards again—using a different criterion—and then to keep on sorting them until you have run out of criteria.

For example, if the task was sorting pictures of different types of house, you might sort them into groups "brick", "stone", "wood", etc., depending on their main material of construction; the second time you might divide the cards into groups called "one", "two" and "three", depending on the number of floors in each building.

In this task, we would like you to concentrate on how programs are constructed, rather than on superficial surface detail. For instance, if you were sorting pictures of houses, you might sort the houses in a variety of ways relating to construction, such as whether they required deep foundations, or whether the brickwork would be complicated, or whether there were internal load-bearing walls, rather than on superficial details such as the colour of the brick.

You are welcome to use any criteria you like, and any groups you like, including "don't know", "not sure" and "not applicable". The main thing is to use only one criterion in each sort—please don't lump two or more in together. If you're not sure about something, just ask.

You may have noticed that the cards are numbered: this is for convenience when recording the results. The numbering is random, so please don't use that as a criterion for sorting!

If you have any comments or questions, then please say, and we will sort them out.

Thank you for your help.

1 function	2 method
3 procedure	4 dependency
5 object	6 decomposition

7 abstraction	8 if-then-else
9 boolean	10 scope
11 list	12 recursion

13 choice	14 state
15 encapsulation	16 parameter
17 variable	18 constant

19 type	20 loop
21 expression	22 tree
23 thread	24 iteration

25	26
array	event

3. Analysis Protocol

For each subject:

Mechanical Analysis

- List all sorts with criteria names (e.g. HS01-1 "Data Structures")
- Count the number of sorts per subject.
- Count the number of categories per criterion.
- Enter sorts into EZSort (`http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/410`)
    - Enter each sort as HS01-1 for the first sort of the first subject, then HS01-2 for the second etc.
    - Name each category

Quantitative Analysis

- Construct a co-occurrence matrix (using Excel spreadsheet and macro provided)
- Using EZCalc, produce a "complete" dendrogram for each subject's set of sorts

Qualitative Analysis

- Note the types of criteria used: conceptual/procedural, subjective/objective, feature driven/abstraction driven

Across all subjects:

- Identify verbatim commonality between criteria (e.g. "Data Structures")
- Identify gist commonality between categories. This will involve looking at correspondence of categories, elaboration with the data gathered in laddering, consultation with your external expert and searching out significant absences. It may also involve comparative examination of the distribution of categories within criteria.

4. Background

Pilot Study

Design of study materials

We generated three sets of stimuli

i. A set of 9, one- or two-paragraph descriptions and examples aimed at capturing key concepts of program structure and construction (as identified by experienced educators) such as: choice (control structures), repetition of actions, delegation of action, state, type, scope, composition of action. The examples took the form either of commented pseudo-code fragments (e.g., the fox, the hen, and the corn problem, an averaging procedure) or of analogous non-computing fragments (e.g., knitting pattern, flowchart for setting an alarm clock, description of shopping list usage).

ii. A set of 10 classic, short programming problems. These were exercises taken from the end-of – chapter examples (2 per chapter) of Shackelford's *Introduction to Computing and Algorithms*, Addison-Wesley 1998

iii. A set of 26 "minimalist", one-word prompts for programming concepts. The prompts were drawn from programming textbooks, from papers on program categorisation, and from lists generated by programming experts and programming educators. The selected set was vetted by two expert programming educators.

We ran pilot studies on two of the sets:

i. The pseudo-code and non-computing fragments were tested with three students and six educators. It was evident that the set was not appropriate, because it was too disparate in form and in abstraction level. The mixture required both domain abstraction and high-level reasoning – as well as substantial comprehension – making it difficult for novices to generate meaningful sorts.

ii. The minimalist set was tested with 7 subjects in two different locations. Three were end-of-first-year CS students of varying backgrounds and ability. Two were computing researchers. Two were expert programmers. The set proved usable by all subjects, viewpoints were offered spontaneously, and the data were manageable. Preliminary cluster analysis suggested possible interesting novice/expert differences, and possible indications of mis-conceptions.

Literature that contributes to the question

Beth Adelson, *Problem Solving And The Development Of Abstract Categories In Programming Languages.* Memory & Cognition 1981 Vol. 9 (4) 422-433

C. M. Allwood, *Novices On The Computer: A Review Of The Literature* International Journal of Man-Machine Studies 1986 Vol 25 633-658

Simon P. Davies, David J. Gilmore, Thomas R. G. Green *Are objects that important? The Effects Of Expertise And Familiarity On The Classification Of Object-Oriented Code* 1995 Human-Computer Interaction Vol. 10 (2&3) 227-248 and
http://www.psychology.nottingham.ac.uk/staff/dg/HCI_paper/HCI_paper.html

Literature that contributes to the methodology

G. Rugg and P. McGeorge *The Sorting Techniques: Card Sorts, Picture Sorts and Item Sorts* Expert Systems 1997 Vol. 17 (2)

G. Rugg *Laddering* 2002

Background literature (optional)

… on Categorization

Eleanor Rosch, Carolyn B. Mervis, Wayne D. Gray, David M. Johnson, Penny Boyes-Braem *Basic Objects in Natural Categories* 1976 Cognitive Psychology Vol. 8: 382-439
Carolyn B. Mervis, Eleanor Rosch *Categorization of Natural Objects* 1981 Annual Review of Psychology 32, 89-115
(and there's an interesting interview with Eleanor Rosch at: http://www.dialogonleadership.org/Rosch-1999.html)
George Lakoff *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind* 1976 University of Chicago Press

… on program conceptualisation

Beth Adelson. *Comparing Natural And Abstract Categories: A Case Study From Computer Science* Cognitive Science 1985 Vol. 9 (4) 417-430
Françoise Détienne *Software Design-Cognitive Aspects* 2001 Springer-Verlag (Practitioner Series)
[Health Warning – great content: terrible translation]
Eliot Soloway and Kate Ehrlich *Empirical Studies of Programming Knowledge* IEEE Transactions on Software Engineering 1984 Vol SE-10 595-609

… on expert/novice distinctions & knowledge consolidation

S. Kaplan, L. Gruppen, L. M. Levanthal and F. Board *The Components of Expertise: a Cross-Disciplinary Review* (University of Michigan, Ann Arbor)
M. W. Eysenck and M. T. Keane *A Handbook of Cognitive Psychology* 3rd edition. 1995 Psychology Press.