## Relating software development in EM to engineering design

The construction of EM models reflects activities that are commonplace in engineering design.

### 1. Design builds on empirical foundations

Systems and products emerge from experiment: the designer begins with an ill-defined notion of what can be achieved, and at first interacts freely in an unconstrained fashion; subsequently, the designer identifies ways of configuring components and managing scenarios for interaction that establish the basis for a prototype system or product. (If the design is routine, the empirical foundations may not need to be made explicit. In radical design, the empirical foundations must necessarily be exposed and explored.)

You can study this sort of activity in relation to the development of the heapsorting technique, as set out in the README file in `~wmb/public/HEAPSORT/HEAPSORT`. This shows how an environment that resembles an electronic blackboard on which an algorithm designer might conceive the idea of the heap structure and of heap maintenance can support a routine pattern of interaction such as is represented in an animation of heapsort.

To see how the initially loose relationship between the computer artefact and its operational interpretation can be transformed into the kind of relationship that is more characterisic of that between a formal specification and a program, alternative versions of heapsort were developed by Jaratsri Rungrattanaubol (see `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/phd/jrungrattanaubol/`). One of these models demonstrates how a WP specification of heapsort can be regarded as a form of high-level observation of the heap artefact, subject to presuming that the interaction with it is highly constrained.

### 2. Experimentation with components

When justifying a design, the engineer appeals to experiments that have been performed on the components of the system or product in isolation from the whole. This typically involves extracting a part from the system (e.g. a strut from a bridge) and placing it in a laboratory environment where it can be tested as if it were in its working context (e.g. providing a test harness to apply forces to the strut).

This kind of activity is in principle rather easy to demonstrate: it involves extracting groups of definitions ("observables") from an EM model, and exercising them in isolation.In practice, this is not so easy to do with current tools. The techniques of isolating parts of a model by extracting definitions from the symboltable database of eden definitions (as developed by J-P Dupont), and inspecting the dependencies between subsets of observables (using the dependency modelling tool developed by Allan Wong) can be combined to illustrate how the inspection of a part of a model may in due course be made much simpler.

Relevant eden files extracted from the draughts model that describe the model of a single square can be found (together with a dmt model to display the observables and their dependencies) in the directory: `~wmb/public/cs405/draughts`. The extracted model contains all the observables referring to a square, together with the environmental elements that are needed to supply the context (for instance, the locations of the white and black pieces, and whether they have been crowned or not). You can test out this model by assigning different values to observables. To place a white or black piece on the square under observation, use a definition of the form

```
w1 = [6,8];
b11 = [6,8];
```

etc., and to model placing pieces off the board, use (e.g.)

```
w1 = [1000,1000];
```

You can experiment by changing the background colour of the squares (see the observables `bgcol` and `bgcolor`) and by crowning pieces (this involves changing the values recorded in the binary array bcrowned, which has 12 entries corresponding the to pieces `b1` through `b12`). In the process of experiment, you will be able to uncover some known 'problems' (or features) of the draughts model relating to changing the colour of squares, and to displaying crowned pieces (stemming from the absence of layering in DoNaLD).

**Some points of difference**

One aspect of engineering that is not (typically) represented in EM activity is the optimisation-to-function once the function of a product or system has been circumscribed. This issue is addressed in the rather limited research that has been done by Simon Yung and several final year project students into extracting conventional programs from EM models. The extract from a PASCAL version of noughts-and-crosses below (see `~wmb/public/projects/games/OXO/PASCAL`) derived by Simon Yung from the tkeden OXO model `oxoJoy1994` illustrates how the dependencies within the OXO model can be 'compiled' into the structure of a PASCAL program (on the assumption that these dependencies are not subject to be redefined). More radical forms of translation of an EM model to a procedural program that are much more highly optimised are of course possible. It is significant that classical computer science reflects a necessary obsession with optimisation stemming from the limitations of early computers. Modelling construals is far more computationally intensive than programming recipes.

*An extract derived from oxoJoy1994 by translation into PASCAL*

```
#include "line.p"
function nofx: integer;

var
    i: integer;
    result: integer;
begin
    result := 0;
    for i := 0 to nofsquares - 1 do
        if board[i] = X then
            result := result + 1;
    nofx := result
end; { nofx }

function full: boolean;
begin
    full := nofo + nofx = nofsquares
end; { full }


function xwon: boolean;
var
    i: integer;
    result: boolean;
    setx: set of SQUARE;
begin
    setx := [];
    for i := 0 to nofsquares - 1 do
        if board[i] = X then
            setx := setx + [i];
    i := 0;
    result := false;
    while (result = false) and (i < lines.noflines) do begin
        if lines.line[i] <= setx then
            result := true;
        i := i + 1
    end;
    xwon := result
end; { xwon }

function draw: boolean;
begin
    draw := not xwon and not owon and full
```

```
end; { draw }
```