1. (a) A basic application of the Model-View-Controller pattern is:

- Develop a core model which is independent of style of input/output.
- Define different views needed for parts/whole model.
- Each view component retrieves data from the model and displays it.
- Each view component has an associated controller component to handle events from users.
- The model component notifies all appropriate view components whenever its data changes.

Advantages include:

- Multiple views of the same model.
- View synchronisation.
- View components can be plugged in.
- Changes to interfaces without touching the model.

Disadvantages include:

- Too complex for simple interface problems.
- Frequent events may strain simple change propagation mechanisms.
- Views and controllers aren't modular in practice.
- Changing the model is expensive if there are many views.

(b) First let's ignore the multiplier, $M$, so $E = C * P^s$

Then we estimate $C$ and $s$ for different types of project:

**Simple** ($E = 2.4 * P^{1.05}$) : A well understood application developed by a small team.

**Intermediate** ($E = 3.0 * P^{1.12}$) : A more complex project for which tem members have limited experience of related systems.

**Embedded** ($E = 3.6 * P^{1.20}$) : A complex project in which the software is part of a complex of hardware, software, regulations and operational constraints.

If we want to use $M$ then it often is estimated as a function of key attributes of the problem. The effect is to modify the basic estimate.

Where do the attributes come from?

- Product attributes (*e.g.* reliability).
- Computer attributes (*e.g.* memory constraints).
- Personnel attributes (*e.g.* programming language experience).
- Project attributes (*e.g.* project development schedule).

For example, if the basic estimate is 1216 person-months.

| Attribute | Magnitude | Multiplier |
|---|---|---|
| Reliability | V high | 1.4 |
| Complexity | V high | 1.3 |
| Memory constraint | High | 1.2 |
| Tool use | Low | 1.1 |
| Schedule | Accelerated | 1.23 |

Then $1216 * 1.4 * 1.3 * 1.2 * 1.1 * 1.23 = 3593$

(c) Fuzzy estimation, at its most basic, requires size classes of code, taken from experience of similar projects.

| Range | Size | Low | High |
|---|---|---|---|
| V Small | 2 | 1 | 4 |
| Small | 8 | 4 | 16 |
| Medium | 32 | 16 | 64 |
| Large | 128 | 64 | 256 |
| V Large | 512 | 256 | 1028 |

We need lots of recent historical data for this and it only gives a gross estimate.

Standard Component Estimating works like this:

- Gather historical data on key components.
- Guess how many of each type you will need ($M_i$).
- Also guess largest ($L_i$) and smallest ($S_i$) extremes.
- Final estimate ($E_i$) is a function of $M_i$, $L_i$ and $S_i$.
- For example, $E_i = (S_i + (4 * M_i) + L_i)/6$

Function Point Estimating is based on a weighted count of common functions of software, typically inputs, outputs, inquiries, data files and interfaces. We may adjust function point total using "influence factors".

(d) For early and regular delivery some advantages are:

- If you hit an unexpected problem you have lost no more than the internal delivery period.
- You may be able to iron out minor problems in the next cycle.
- A way of gathering solid data early.
- Boosts morale if releases make progress.

but if we over-indulge the consequent problems include:

- Cycle too fast and setup/test may eat up your time.
- Need effort to monitor fast cycles.
- Saps morale if releases are chaotic.

Advantages of explicit ownership of components and functionality include:

- Ownership and consistency of functions, not just components.
- Assists sharing of components across teams.

but there are overdose effects, including:

- Possible friction between function and component owners.
- Can become lost in the interconnections between functions and components.
- Ownership by function turns components into shared areas.
- Ownership by components turns functions into orphan areas.

2. (a) A basic application of the Blackboard pattern is:

    - Problem solvers work independently (and opportunistically) on parts of the problem.
    - Share a common data structure (the blackboard).
    - A central controller manages access to the blackboard.
    - The blackboard may be structured (*e.g.* into levels of abstraction) so problem solvers may work at different levels.

    Advantages include:

    - Easy (within limits) to experiment with different problem solvers and control heuristics.
    - Knowledge sources may be re-usable.
    - System may (within limits) be tolerant to dodgy problem solvers by protecting/revising the blackboard.

    Disadvantages include:

    - Difficult to test.
    - Difficult to guarantee an optimum solution.
    - Control strategy often heuristic.
    - May be computationally expensive.
    - Parallelism possible but in practice we need much synchronisation.

   (b) There are various development models: the "dictatorship" model used by Linux with a single leader at the top of a hierarchy of top-level developers and sub-developers; the decentralised model used by Apache with a team of core developers voting on major decisions and coordinated by release managers; or the "organisational dictatorship" of Mozilla which resembles Linux but with the Mozilla organisation rather than Linus Torvalds at the top of the hierarchy.

    Potential advantages include: defusing of piracy issue; more extensive code review; a committed developer base; and lower development overheads. Disadvantages include: difficulty in generating enough cask from open source to pay for support, documentation, *etc.*; less interesting tasks get less attention; release cycles can be slow and unpredictable; assumes a high level of general technical competence.

   (c) From the lectures, students should remember the equation below, although any other properly explained function would do.

$$y(N) = \frac{r(N)}{r(N) + e(N)}$$

    where:

- $y(N)$ is fraction of defects removed in step $N$
- $r(N)$ is the number of defects removed at step $N$.
- $e(N)$ is the number of defects escaping at step $N$.

The difficulty with this equation is that we can only estimate $e(N)$ as a function of $e(1), \ldots, e(N-1)$.

(d) Potential problems include:

- These may be big components: tempting but the most difficult to re-use.
- Re-used components are older so may reach obsolescence sooner.
- May not be able to re-use component and documentation.
- May be hard to find the original designer if it goes wrong.
- Tempting to twist project to fit re-usable components.
- Should have considered re-use in the *previous* project - it may be too late now.

3. (a) A basic application of the Layers pattern is:

- Start with the lowest level.
- Build higher layers on top.
- Same level of abstraction within a layer.
- No component spreads over two layers.
- Try to keep lower layers leaner.
- Specify interfaces for each layer.
- Try to handle errors at lowest layer possible.

Advantages include:

- Reuse of layers.
- Standardisation of tasks and interfaces.
- Local dependencies between layers.

Disadvantages include:

- Cascades of changing behavior.
- Lower efficiency of data transfer.
- Difficult to choose granularity of layers.

(b) Find all the paths through this flow graph:

- 1,2,3,4,12,13
- 1,2,3,5,6,11,2,12,13
- 1,2,3,5,7,8,10,11,2,12,13
- 1,2,3,5,7,9,10,11,2,12,13

If we follow all these paths we know:

- Every statement in the routine has been executed at least once.
- Every branch has been exercised for a true/false condition.

(c) Examples of measures, from the lecture notes, are given below. One from each table is sufficient if accompanied by appropriate explanation.

Schedule:

| Category | Measure |
|---|---|
| Milestone | Date of delivery |
| Work unit | Component status<br>Requirement status<br>Paths tested<br>Problem report status<br>Reviews completed<br>Change request status |

Cost:

| Category | Measure |
|---|---|
| Personnel | Effort<br>Staff experience<br>Staff turnover |
| Financial performance | Earned value<br><br>Cost |
| Environment availability | Availability dates<br><br>Resource utilisation |

Growth:

| Category | Measure |
|---|---|
| Product size and stability | Lines of code<br><br>Components<br>Words of memory<br>Database size |
| Functional size and stability | Requirements<br><br><br>Function points<br>Change request workload |

Quality:

| Category | Measure |
|---|---|
| Defects | Problem reports<br>Defect density<br>Failure interval |
| Rework | Rework size<br>Rework effort |

Ability:

| Category | Measure |
|---|---|
| Process maturity | Capability maturity model level |
| Productivity | Product size/effort<br>Functional size/effort |

Technology:

| Category | Measure |
|---|---|
| Performance | Cycle time |
| Resource utilisation | CPU utilisation<br><br>I/O utilisation<br>Memory utilisation<br>Response time |

(d) There are numerous arguments agains this. One quite precise argument from the lectures is given below, although less formal arguments are acceptable.

Assuming we start with no defects, because we were tough on defect detection in the early coding. Roughly:

$$P_i = (1 - p)^i$$

where:

- $p$ is the probability of injecting a defect at a stage.
- $i$ is the number of stages.
- $P$ is the probability of a defect-free product at stage $i$.

$$0.904 = (1 - 0.01)^{10}$$
$$0.4057 = 0.5 * (1 - 0.01)^9$$

so a high probability of fault injection in one step radically drops the overall probability of freedom from defects.
On the other hand:

$$R_i = N * (1 - y)^i$$

where:

- $N$ is the initial number of defects.
- $y$ the fraction of defects removed per stage.
- $i$ is the number of stages.
- $R_i$ is the number of defects remaining at stage $i$.

$$32 = 100000 * (1 - 0.8)^5$$
$$96 = 100000 * (1 - 0.4) * (1 - 0.8)^4$$

so dropping a lot lower on one stage of a high quality defect removal process has a small effect on overall yield.
If anything close to the above relationships apply then it is better to be close to defect-free than to rely on fixes.