Unit II

4 Selection of an Appropriate Project Approach

Learning Objective:

- Building versus buying software
- Taking account of the characteristics of the project
- Process models
  - Waterfall
  - Prototyping and iterative approaches
  - Incremental delivery
- Agile approaches

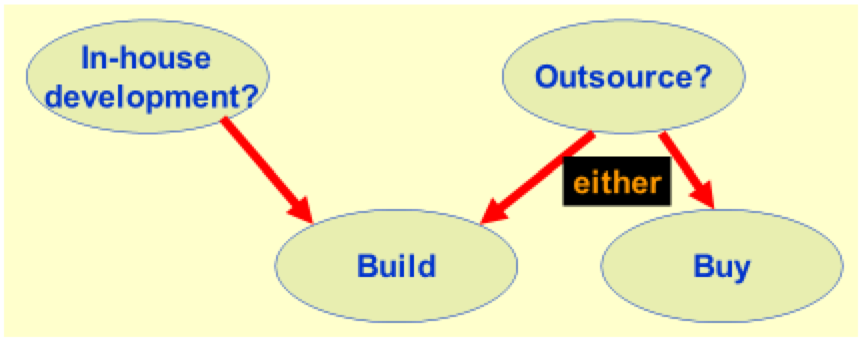
It can be argued that agile approaches are often a repackaging of prototyping and incremental delivery.

Selection of project approaches

- This part concerned with choosing the right approach to a particular project: variously called *technical planning*, *project analysis*, *methods engineering* and *methods tailoring*
- In-house: often the methods to be used dictated by organizational standards
- Suppliers: need for tailoring as different customers have different needs

Section 4.1 and 4.3 of the textbook discuss these issues. Different types of project need different types of approach. If you are working in one particular environment which specializes in one type of software, then the approach is likely not to change much from one project to another. Where you work *for different clients in different organizations developing a variety of applications then the approach for each project may need to be tailored.*

Build or buy?



In-house development almost always involves developing new code.

If it is decided to use a specialist organization to implement system, the supplier could either build a 'bespoke' system for you, or could install a pre-existing application. There are hybrids of these options, e.g. to build in-house but use temporary contract staff, or Customised Off-the-Shelf (COTS) where a basic existing core system is modified for your particular requirements.

Some advantages of off-the-shelf (OTS) software

- Cheaper as supplier can spread development costs over a large number of customers
- Software already exists
    - Can be trialled by potential customer
    - No delay while software being developed
- Where there have been existing users, bugs are likely to have been found and eradicated

Some possible disadvantages of off-the-shelf

- Customer will have same application as everyone else: no competitive advantage, *but* competitive advantage may come from the *way* application is used
- Customer may need to change the way they work in order to fit in with OTS application
- Customer does not own the code and cannot change it
- Danger of over-reliance on a single supplier

One concern is what happens if the supplier goes out of business. Customer might not then be able to maintain system: hence the use of 'escrow' services where a 3rd party retains a copy of the code.

General approach

- Look at risks and uncertainties e.g.
    - are requirement well understood?
    - are technologies to be used well understood?
- Look at the type of application being built e.g.
    - information system? embedded system?
    - criticality? differences between target and development environments?
- Clients' own requirements
    - need to use a particular method

See section 4.3 of the textbook.

Structure versus speed of delivery

Structured approach

- Also called 'heavyweight' approaches
- Step-by-step methods where each step and intermediate product is carefully defined
- Emphasis on getting quality right first time
- Example: use of UML and USDP
- Future vision: Model-Driven Architecture (MDA). UML supplemented with Object Constraint Language, press the button and application code generated from the UML/OCL model

Agile methods

- Emphasis on speed of delivery rather than documentation
- RAD Rapid application development emphasized use of quickly developed prototypes

- JAD Joint application development. Requirements are identified and agreed in intensive workshops with users

Processes versus Process Models

- Starting from the inception stage:
  - A product undergoes a series of transformations through a few identifiable stages
  - Until it is fully developed and released to the customer.
  - This forms its life cycle or development process.
- Life cycle model (also called a process model):
  - A graphical or textual representation of the life cycle.

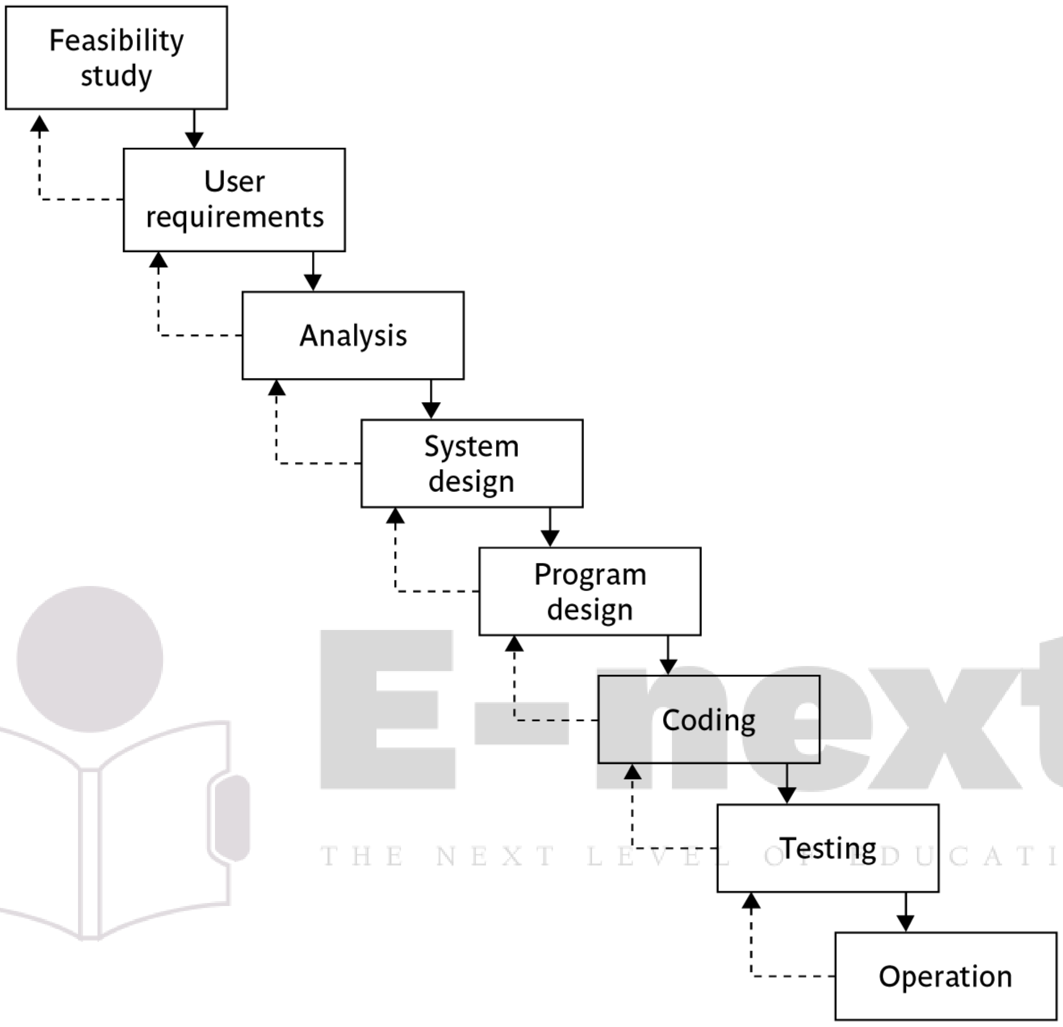
Choice of process models

- 'waterfall' also known as 'one-shot', 'once-through'
- incremental delivery
- evolutionary development

Also use of 'agile methods' e.g. extreme programming

It could be argued that extreme programming is mainly a particular way of carrying out incremental and evolutionary development

Waterfall



Section 4.6 of the textbook discusses this topic.

We have already touched upon the Waterfall approach in Chapter 1 when we discussed the ISO 12207. The way that the Waterfall approach is usually interpreted, it implies that all work on one phase has to be completed and checked off before the next one can start. However, it can be argued that ISO 12207 is really a technical model showing the order technical processes need to be carried out on a software component. You might break down an application into component increments, but the technical processes relating to that increment are carried out in the ISO 12207 sequence. You can, within the ISO 12207 sequence, loop back in an iterative manner, but the technical sequence still remains.

- the 'classical' model
- imposes structure on the project
- every stage needs to be checked and signed off

- BUT
    - limited scope for iteration
- V model approach is an extension of waterfall where different testing phases are identified which check the quality of different development phases

Evolutionary delivery: prototyping

‘ *An iterative process of creating quickly and inexpensively live and working models to test out requirements and assumptions’*

Sprague and McNurlin

main types

- ‘throw away’ prototypes
- evolutionary prototypes

what is being prototyped?

- human-computer interface
- functionality

Reasons for prototyping

- learning by doing
- improved communication
- improved user involvement
- a feedback loop is established
- reduces the need for documentation
- reduces maintenance costs i.e. changes after the application goes live
- prototype can be used for producing expected results

- *A prototype is a working model of one or more aspects of the project application. It is constructed quickly and inexpensively in order to test out assumptions.*

- *Sections 4.9, 4.10 and 4.11 discuss this topic.*
- *learning by doing - useful where requirements are only partially known*
- *improved communication - users are reluctant to read massive documents, but when system is 'live' you get a better feeling for it*
- *improved user involvement - user ideas and requests are quickly implemented*
- *the reduction of maintenance costs – the idea is that if you do not have a prototype then the first release of the application will effectively become a prototype as users will find things that they do not like and then ask for them to be changed. It is easier and safer to make such changes before the application becomes operational.*
- *testing - involves devising test cases and then documenting the results expected when the test cases are run. If these are done by hand, then effectively you have a manual prototype. Why not get the software prototype to generate the expected results?*

prototyping: some dangers

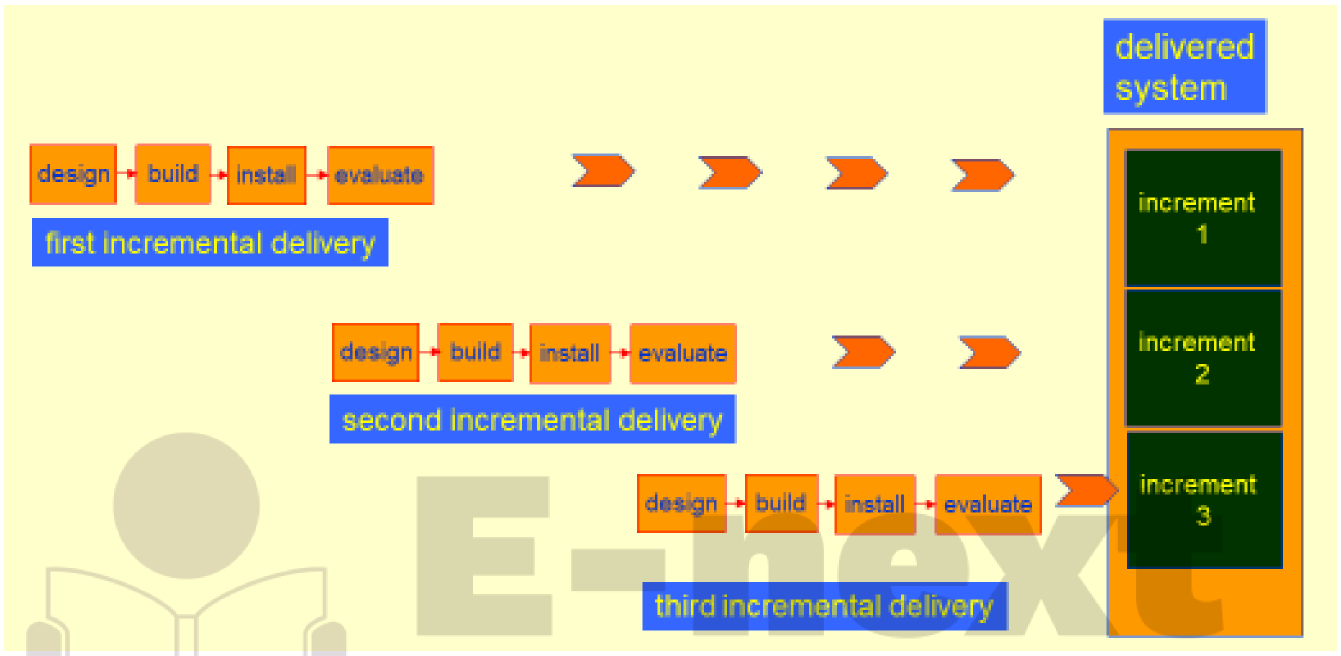
- users may misunderstand the role of the prototype
- lack of project control and standards possible
- additional expense of building prototype
- focus on user-friendly interface could be at expense of machine efficiency

other ways of categorizing prototyping

- what is being learnt?
  - organizational prototype
  - hardware/software prototype ('experimental')
  - application prototype ('exploratory')
- to what extent
  - mock-ups
  - simulated interaction
  - partial working models: *vertical* versus *horizontal*

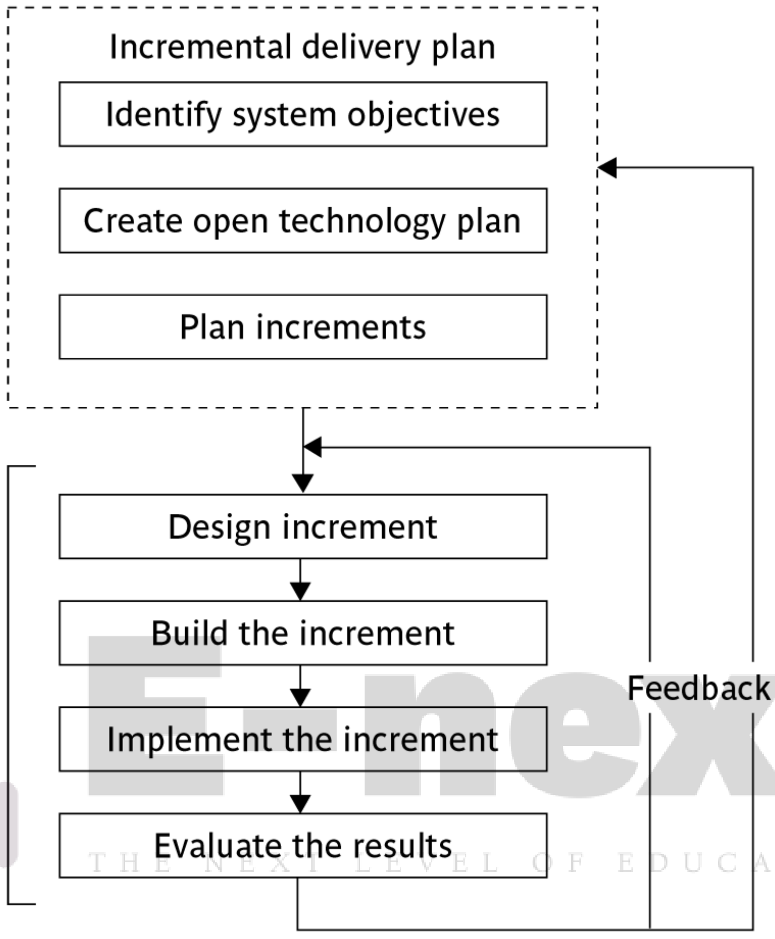
When a prototype is to be produced as part of a student's final year project, then the learning objectives that the prototypes will help be achieved need to defined at project inception. The details of *what has been learnt through the development and exercising of the prototype should be documented during the project.*

Incremental delivery


delivered system
design
build
install
evaluate
first incremental delivery
second incremental delivery
third incremental delivery
increment 1
increment 2
increment 3

The application to be delivered is broken down into a number of components each of which will be actually used by the users. Each of these is then developed as a separate 'mini-project' or increment.

The incremental process


Incremental delivery plan
Identify system objectives
Create open technology plan
Plan increments
Design increment
Build the increment
Implement the increment
Evaluate the results
Repeat for each increment
Feedback

This approach has had a very vocal advocate in Tom Gilb (see the book Principles of Software Engineering Management published by Addison-Wesley (1988)).

Gilb argues that the initial focus should be on high level business objectives. There may be many ways in which important objectives can be achieved and we ought to allow ourselves freedom in the way we try to achieve the objectives.

Open technology plan – we need to ensure that the technologies we use are ones that facilitate the addition of components to an existing application.

Plan increments – the nature and order of the increments to be delivered needs to be delivered to the users have to be planned at the outset.

Incremental approach: benefits

- feedback from early stages used in developing latter stages
- shorter development thresholds
- user gets some benefits earlier
- project may be put aside temporarily
- reduces ‘gold-plating’

BUT there are some possible disadvantages

- loss of economy of scale
- ‘software breakage’

Advantages of prototyping

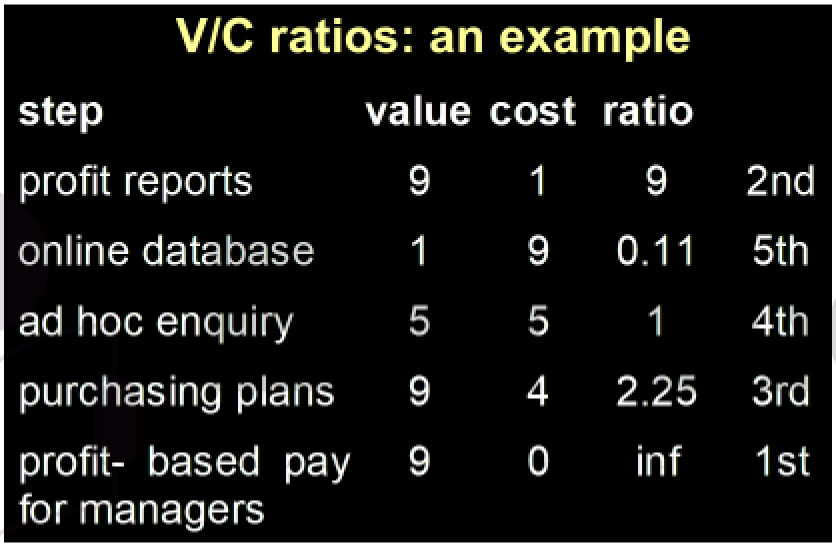
- feedback from early stages used in developing latter stages
- shorter development thresholds - important when requirements are likely to change
- user gets some benefits earlier - may assist cash flow
- project may be put aside temporarily - more urgent jobs may emerge
- reduces ‘gold-plating’ i.e. features requested but not used
- *But there are possible disadvantages*
- loss of economy of scale - some costs will be repeated
- ‘software breakage’ - later increments might change earlier increments

Overview of incremental plan

- steps ideally 1% to 5% of the total project
- non-computer steps should be included
- ideal if a step takes one month or less:
    - not more than three months
- each step should deliver some benefit to the user
- some steps will be physically dependent on others

which step first?

- some steps will be pre-requisite because of physical dependencies
- others may be in any order
- value to cost ratios may be used
    - V/C where
    - V is a score 1-10 representing value to customer
    - C is a score 0-10 representing value to developers



V/C ratios: an example

step	value	cost	ratio	
profit reports	9	1	9	2nd
online database	1	9	0.11	5th
ad hoc enquiry	5	5	1	4th
purchasing plans	9	4	2.25	3rd
profit- based pay for managers	9	0	inf	1st

Genesis of 'Agile' methods

Structured development methods have several disadvantages

- produce large amounts of documentation which can largely remain unread
- documentation has to be kept up to date
- division into specialist groups and need to follow procedures stifles communication
- users can be excluded from decision process
- long lead times to deliver anything etc. etc

The answer? 'Agile' methods?

Agile Methods

- Agile is an umbrella term that refers to a group of development processes:
  - Crystal technologies
  - Atern (formerly DSDM)
  - Feature-driven development
  - Scrum
  - Extreme Programming (XP)
- Similar themes:
  - Some variations

Important Themes of Agile Methods

- Base on the incremental approach:
  - At a time, only one increment is planned, developed, and then deployed at the customer site.
- Agile model emphasizes face-to-face communication over written documents.
- An agile project usually includes a customer representative in the team.
- Agile development projects usually deploy pair programming.

Atern/Dynamic system development method (DSDM)

- UK-based consortium
- *arguably* DSDM can be seen as replacement for SSADM
- DSDM is more a project management approach than a development approach
- Can still use DFDs, LDSs etc!
- An update of DSDM has been badged as ‘Atern’

A fuller explanation can be found in the DSDM Atern Pocket Book published by the Atern/DSDM Consortium.

SSADM is Structured Systems Analysis and Design Method a very heavy-weight and bureaucratic methodology that was promoted by the UK government

DFD = Data Flow Diagram

LDS = Logical Data Structure, effectively an Entity-Relationship Diagram

Six core Atern/DSDM principles

1. Focus on business need
2. Delivery on time – use of time-boxing
3. Collaborate
4. Never compromise qualitiy
5. Deliver iteratively
6. Build incrementally

This can be seen as a re-packaging of a lot of the ideas that have been discussed under the incremental and evolutionary approaches

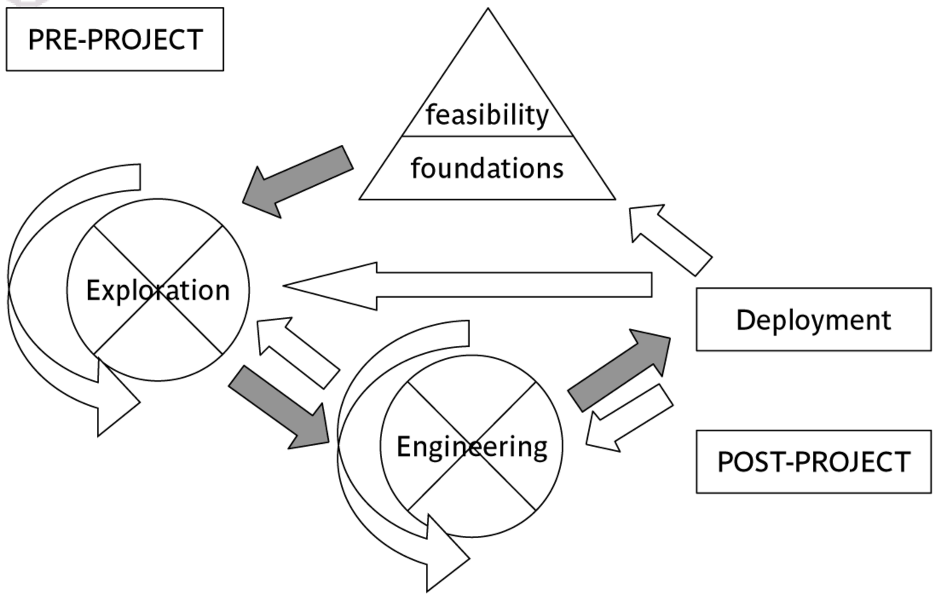


Fig. Atern/DSDM framework

The feasibility/business study stage will not only look at the business feasibility of the proposed project, but also as whether DSDM would be the best framework for it. Applications where there is a prominent user interface would be prime candidates.

Atern DSDM: time-boxing

- *time-box* fixed deadline by which *something* has to be delivered
- typically two to six weeks
- MOSCOW priorities
  - Must have - essential
  - Should have - very important, but system could operate without
  - Could have
  - Want - but probably won't get!

- Time-boxes mean that the focus moves from having a fixed set of functional requirements and then extending the planned project completion until they have all been developed. The deadline is fixed and we deliver what we have completed so far even if it is not everything that was originally planned.
- In order to make the delivered package coherent and as useful as possible to the users, requirements are prioritized according to the MoSCoW rules.

Extreme Programming Model

- Extreme programming (XP) was proposed by Kent Beck in 1999.
- The methodology got its name from the fact that:
  - Recommends taking the best practices to extreme levels.
  - If something is good, why not do it all the time.

Extreme programming

- increments of one to three weeks
  - customer can suggest improvement at any point

- argued that distinction between design and building of software are artificial
- code to be developed to meet current needs only
- frequent re-factoring to keep code structured

- *Associated with Kent Beck –*
- *Developed originally on Chrysler C3 payroll (Smalltalk) project*
- *Agile methods include Jim Highsmith's Adaptive Software Development and Alistair Cocburn's Chrystal Lite methods*

Taking Good Practices to Extreme

- If code review is good:
  - Always review --- pair programming
- If testing is good:
  - Continually write and execute test cases --- test-driven development
- If incremental development is good:
  - Come up with new increments every few days
- If simplicity is good:
  - Create the simplest design that will support only the currently required functionality.
- If design is good,
  - everybody will design daily (refactoring)
- If architecture is important,
  - everybody will work at defining and refining the architecture (metaphor)
- If integration testing is important,
  - build and integrate test several times a day (continuous integration)
- developers work in pairs
- test cases and expected results devised *before* software design

- after testing of increment, test cases added to a consolidated set of test cases

Limitations of extreme programming

- Reliance on availability of high quality developers
- Dependence on personal knowledge – after development knowledge of software may decay making future development less easy
- Rationale for decisions may be lost e.g. which test case checks a particular requirement
- Reuse of existing code less likely

Grady Booch's concern

Booch, an OO authority, is concerned that with requirements driven projects:

'Conceptual integrity sometimes suffers because this is little motivation to deal with scalability, extensibility, portability, or reusability beyond what any vague requirement might imply'

Tendency towards a large number of discrete functions with little common infrastructure?

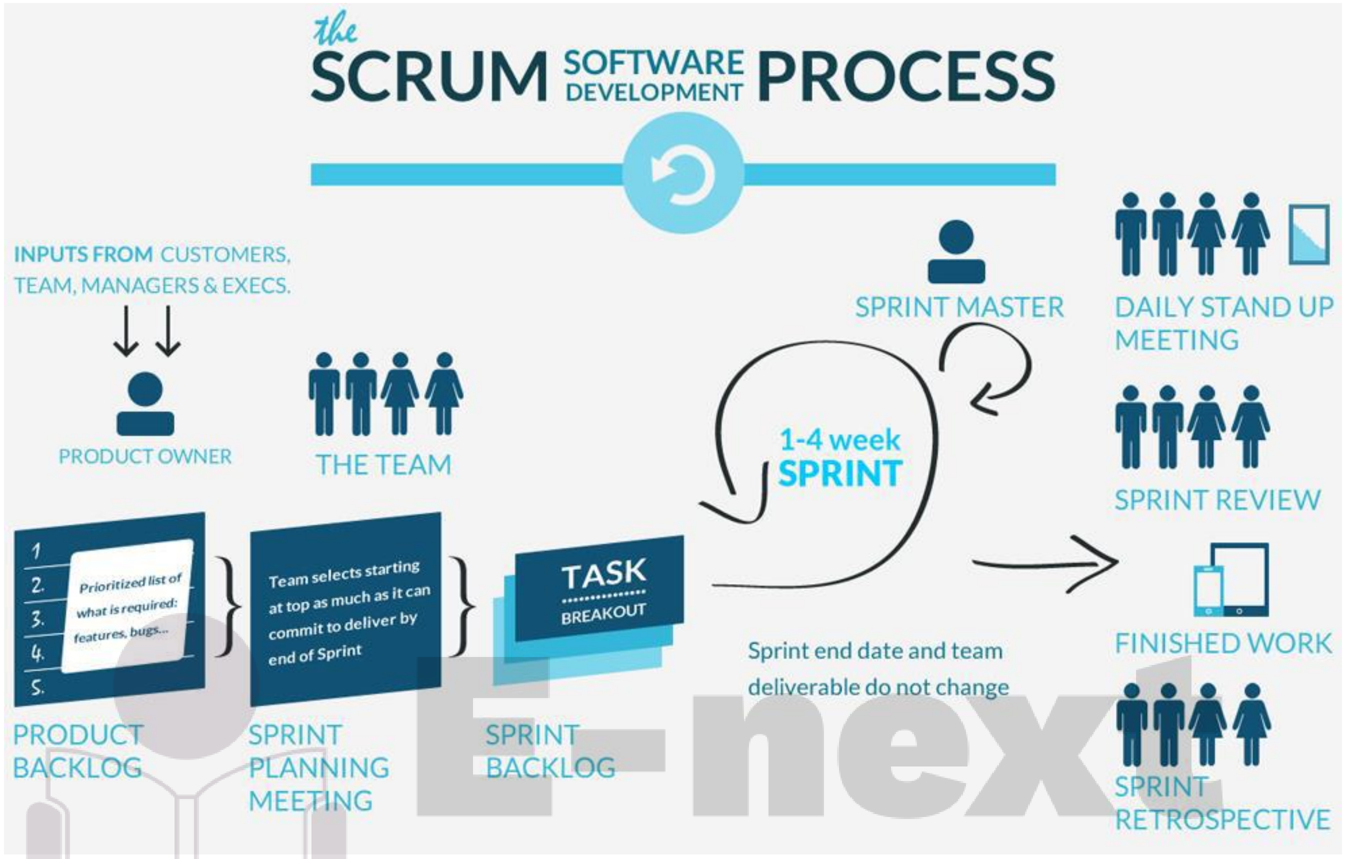
Project Characteristics that Suggest Suitability of Extreme Programming

- Projects involving new technology or research projects.
    - In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.
- Small projects:
    - These are easily developed using extreme programming.

<u>Scrum</u>

- One of the "agile processes"
- Self-organizing teams
- Product progresses in a series of month-long "sprints"

- Requirements are captured as items in a list of “product backlog”


the SCRUM SOFTWARE DEVELOPMENT PROCESS
INPUTS FROM CUSTOMERS, TEAM, MANAGERS & EXECS.
PRODUCT OWNER
THE TEAM
SPRINT MASTER
DAILY STAND UP MEETING
1-4 week SPRINT
SPRINT REVIEW
1. 2. 3. 4. 5.
Prioritized list of what is required: features, bugs...
Team selects starting at top as much as it can commit to deliver by end of Sprint
TASK BREAKOUT
FINISHED WORK
Sprint end date and team deliverable do not change
PRODUCT BACKLOG
SPRINT PLANNING MEETING
SPRINT BACKLOG
SPRINT RETROSPECTIVE

Sprints

- Scrum projects make progress in a series of “sprints”
- Analogous to XP iterations
- Target duration is one month
  - +/- a week or two
- During a sprint, a product increment is designed, coded, and tested

Key Roles in Scrum Process

- Product Owner
  - Acts on behalf of customers to represent their interests.
- Development Team

- Team of five-nine people with cross-functional skill sets.

- Scrum Master
  - Facilitates scrum process and resolves impediments at the team and organization level by acting as a buffer between the team and outside interference.

Scrum Ceremonies

- Sprint Planning Meeting
- Sprint
- Daily Scrum
- Sprint Review Meeting

Sprint Planning

- In this meeting, the product owner and the team members decide which Backlog Items the Team will work on in the next sprint
- Scrum Master should ensure that the Team agrees to realistic goals

Sprint

- Fundamental process flow of Scrum
- A month-long iteration, during which an incremental product functionality completed
- NO outside influence can interfere with the Scrum team during the Sprint
- Each Sprint begins with the Daily Scrum Meeting

Daily Scrum

Held daily:

- Short meeting
- Lasts for about 15mins only

Main objective is to answer three questions:

- What did you do yesterday
- What will you do today?

- What obstacles are in your way?

Sprint Review Meeting

- Team presents what it accomplished during the sprint
    - Typically takes the form of a demo of new features or underlying architecture
- Informal meeting:
    - The preparation time should not exceed about 2-hours

Sprint Artefacts

- Product backlog
- Sprint backlog
- Sprint burndown chart

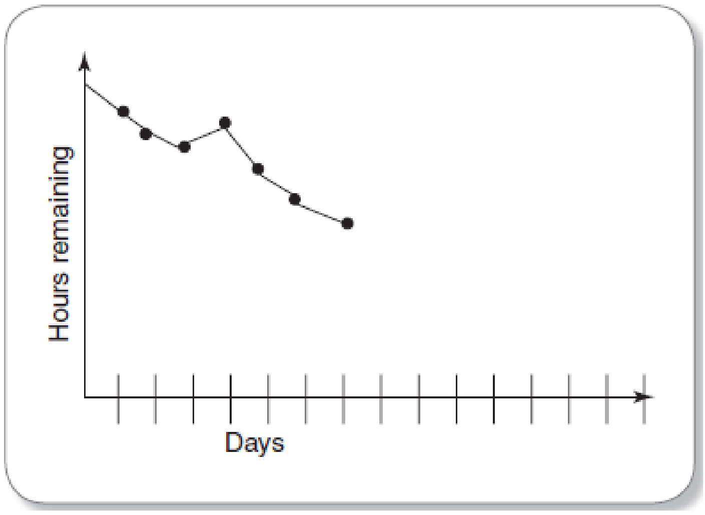
Product Backlog

- A list of all desired work on the project --- usually a combination of :
    - story-based work (e.g. "let user search and replace")
    - task-based work ("improve exception handling")
- List is prioritized by the Product Owner

Sprint Backlog

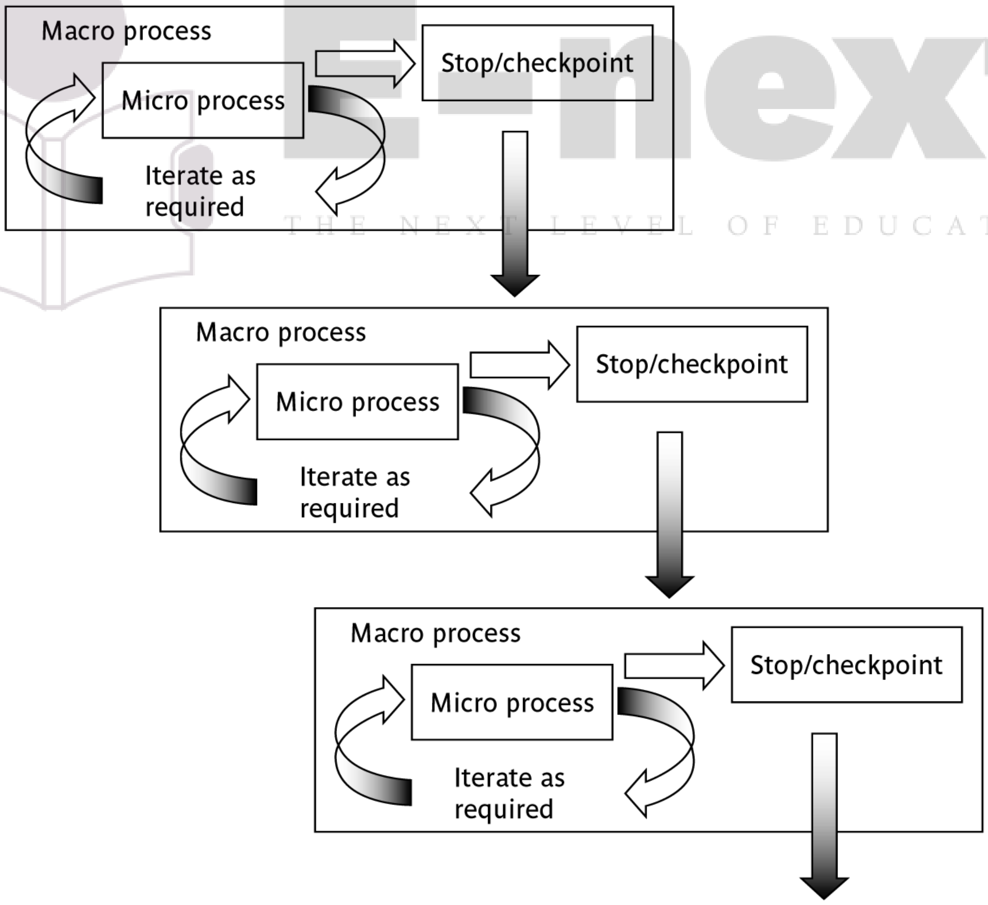
- A subset of Product Backlog Items, which define the work for a Sprint
    - Created by Team members
    - Each Item has it's own status
    - Updated daily

Sprint Burndown Chart



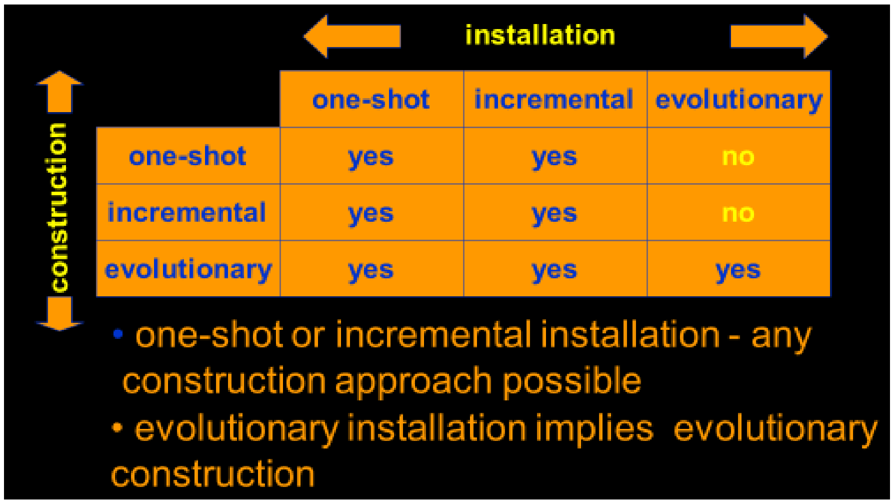
- Day-wise depicts the total Sprint Backlog hours remaining
- Ideally should burn down to zero to the end of the Sprint

macro and micro processes



Section 4.14 discusses these ideas in a little more detail

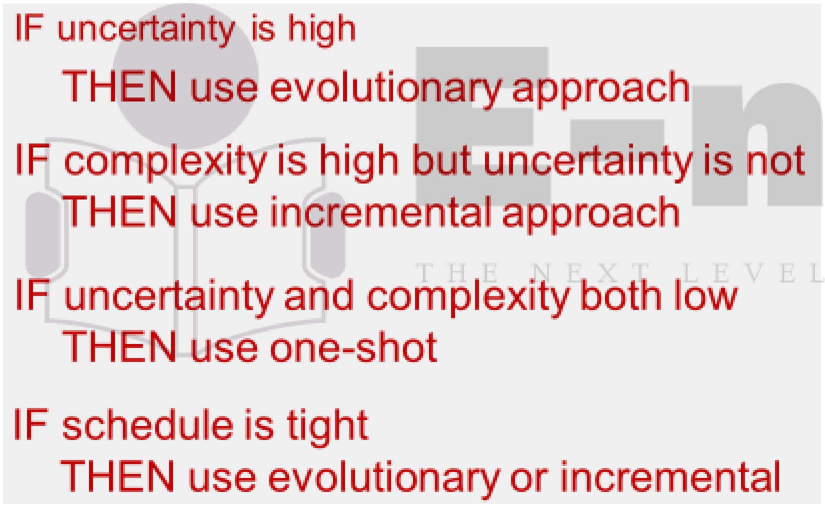
combinations of approach



construction \ installation	one-shot	incremental	evolutionary
one-shot	yes	yes	no
incremental	yes	yes	no
evolutionary	yes	yes	yes

- one-shot or incremental installation - any construction approach possible
- evolutionary installation implies evolutionary construction

'rules of thumb' about approach to be used



IF uncertainty is high
THEN use evolutionary approach

IF complexity is high but uncertainty is not
THEN use incremental approach

IF uncertainty and complexity both low
THEN use one-shot

IF schedule is tight
THEN use evolutionary or incremental

5 Software Effort Estimation

Learning Objectives:

- why estimating is problematic (or ‘challenging’)
- the main generic approaches to estimating, including:
    - Bottom-up versus top-down estimating
    - Parametric models
    - Estimating by analogy
- With regard to parametric models, some particularly well-known methods, namely function points and COCOMO are then discussed in a little more detail. However, the aim is to provide an overview of the principles, not detailed counting rules.

What makes a successful project?

Delivering:	**Stages:**
• agreed functionality • on time at the agreed cost • with the required quality	1. Set targets 2. Attempt to achieve targets

BUT what if the targets are not achievable?

A key point here is that developers may in fact be very competent, but incorrect estimates leading to unachievable targets will lead to extreme customer dissatisfaction.

Some problems with estimating

- Subjective nature of much of estimating
  - It may be difficult to produce evidence to support your precise target
- Political pressures
  - Managers may wish to reduce estimated costs in order to win support for acceptance of a project proposal
- Changing technologies
  - these bring uncertainties, especially in the early days when there is a 'learning curve'
- Projects differ
  - Experience on one project may not be applicable to another

- *Section 5.1. of the textbook.*
- *Exercise 5.1 where the reader is asked to calculate productivity rates for activities on a real project illustrates some of the difficulties of interpreting project data.*

Over and under-estimating

• Parkinson's Law: 'Work expands to fill the time available' • An over-estimate is likely to cause project to take longer than it would otherwise	• Weinberg's Zeroth Law of reliability: 'a software project that does not have to meet a reliability requirement can meet any other requirement'

The answer to the problem of over-optimistic estimates might seem to be to pad out all estimates, but this itself can lead to problems. You might miss out to the competition who could underbid you, if you were tendering for work. Generous estimates also tend to lead to reductions in productivity. On the other hand, having aggressive targets in order to increase productivity could lead to poorer product quality.

Note that 'zeroth' is what comes before first.

This is discussed in Section 5.3 of the text which also covers Brooks' Law.

Basis for successful estimating

- Information about past projects
  - Need to collect performance details about past project: how big were they? How much effort/time did they need?
- Need to be able to measure the amount of work involved
  - Traditional size measurement for software is 'lines of code' – but this can have problems

Despite our reservation about past project data –we still need to collect and analyse it! If we don't know how big the job is, we can't estimate how long it will take to do.

A taxonomy of estimating methods

- Bottom-up - activity based, analytical
- Parametric or algorithmic models e.g. function points
- Expert opinion - just guessing?
- Analogy - case-based, comparative
- Parkinson and 'price to win'

- *This taxonomy is based loosely on Barry Boehm's in the big blue book, 'Software Engineering Economics'.*
- *One problem is that different people call these approaches by different names. In the case of bottom-up and analogy some of the alternative nomenclatures have been listed.*
- *'Parkinson' is setting a target based on the amount of staff effort you happen to have available at the time. 'Price to win' is setting a target that is likely to win business when tendering for work. Boehm is scathing about these as methods of estimating. However, sometimes you might have to start with an estimate of an acceptable cost and then set the scope of the project and the application to be built to be within that cost.*
- *This is discussed in Section 5.5 of the textbook.*

Parameters to be Estimated

- Size is a fundamental measure of work
- Based on the estimated size, two parameters are estimated:
  - Effort

- Duration
- Effort is measured in person-months:
  - One person-month is the effort an individual can typically put in a month.

Person-Month

- Suppose a project is estimated to take 300 person-months to develop:
  - Is one person working for 30 days same as 30 persons working for 1 day?
  - Yes/No? why?
- How many hours is a man month?
  - Default Value: 152 hours per month
  - 19 days at 8 hours per day.

Mythical Man-Month

- "Cost varies as product of men and months, progress does not."
  - Hence the man-month as a unit for measuring the size of job is a dangerous and deceptive myth.
- The myth of additional manpower
  - Brooks Law: "Adding manpower to a late project makes it later"

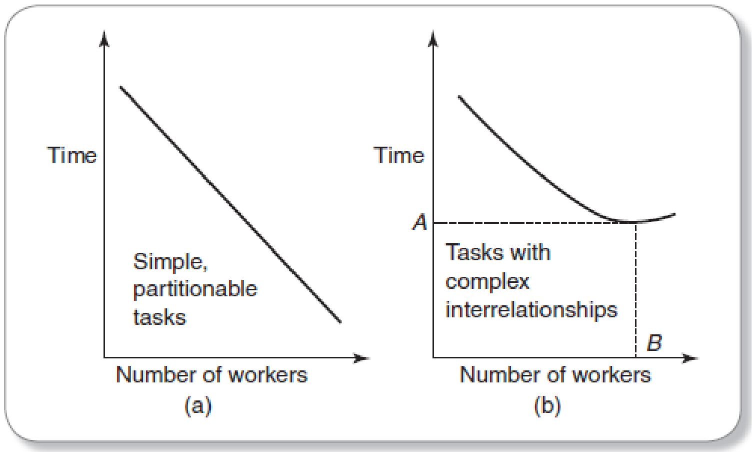


FIGURE 5.2 Impact of addition of workers on the completion time for various types of projects

For tasks with complex interrelationship, addition of manpower to a late project does not help.

Measure of Work

- The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.
- Two metrics are used to measure project size:
    - Source Lines of Code (SLOC)
    - Function point (FP)
- FP is now-a-days favoured over SLOC:
    - Because of the many shortcomings of SLOC.

Major Shortcomings of SLOC

- Difficult to estimate at start of a project
- Only a code measure
- Programmer-dependent
- Does not consider code complexity

Bottom-up versus top-down

- Bottom-up
    - use when no past project data
    - identify all tasks that have to be done – so quite time-consuming
    - use when you have no data about similar past projects
- Top-down
    - produce overall estimate based on project cost drivers
    - based on past project data
    - divide overall estimate between jobs to be done

There is often confusion between the two approaches as the first part of the bottom-up approach is a top-down analysis of the tasks to be done, followed by the bottom-up adding up of effort for all the work to be done.

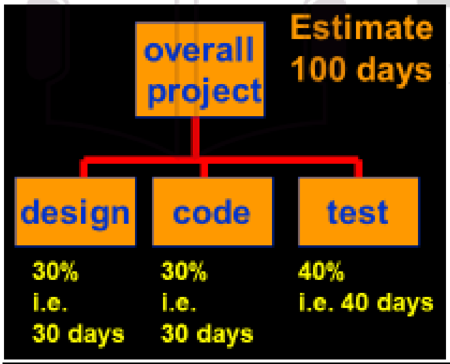
Bottom-up estimating

1. Break project into smaller and smaller components
2. Stop when you get to what one person can do in one/two weeks
3. Estimate costs for the lowest level activities
4. At each higher level calculate estimate by adding estimates for lower levels

The idea is that even if you have never done something before you can imagine what you could do in about a week.

Exercise 5.3 relates to bottom-up estimating

Top-down estimates

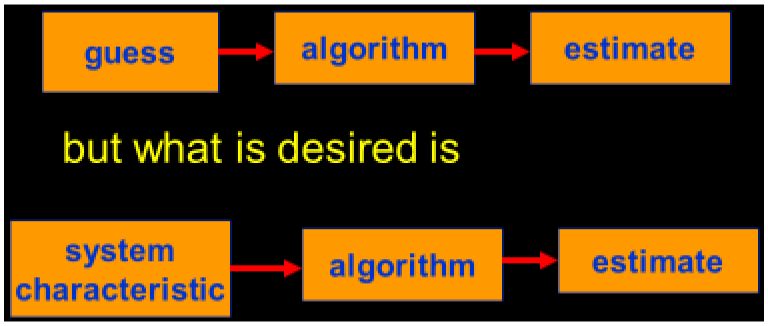


- Produce overall estimate using effort driver(s)
- distribute proportions of overall estimate to components

The initial overall estimate might have been produced using a parametric model, by analogy or just expert judgement.

Algorithmic/Parametric models

- COCOMO (lines of code) and function points examples of these
- Problem with COCOMO etc:



The problems with COCOMO is that the input parameter for system size is an estimate of lines of code. This is going to have to be an estimate at the beginning of the project.

Function points, as will be seen, counts various features of the logical design of an information system and produced an index number which reflects the amount of information processing it will have to carry out. This can be crudely equated to the amount of code it will need.

Parametric models - the need for historical data

- simplistic model for an estimate

  estimated effort = (system size) / productivity

- e.g.

  system size = lines of code

  productivity = lines of code per day

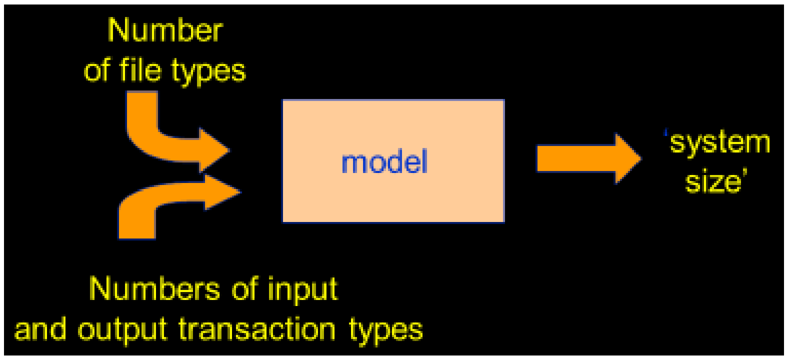
- productivity = (system size) / effort
  - based on past projects

This is analogous to calculating speed from distance and time.

Parametric models

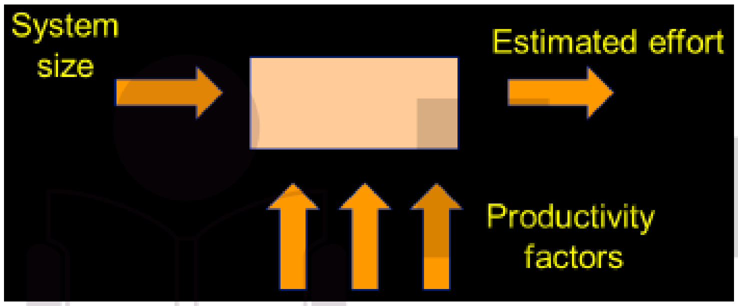
Some models focus on task or system size e.g. Function Points

FPs originally used to estimate Lines of Code, rather than effort



'System size' here can be seen as an index that allows the size of different applications to be compared. It will usually correlate to the number of lines of code required.

- Other models focus on productivity: e.g. COCOMO
- Lines of code (or FPs etc) an input

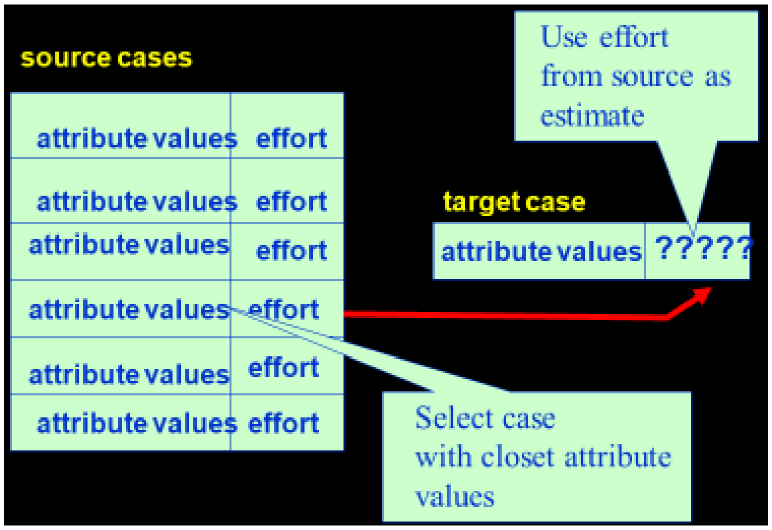


COCOMO originally was based on a size parameter of lines of code (actually 'thousand of delivered sourcecode instructions' or kdsi). Newer versions recognize the use of functions points as a size measure, but convert them to a number called 'equivalent lines of code (eloc).

Expert judgement

- Asking someone who is familiar with and knowledgeable about the application area and the technologies to provide an estimate
- Particularly appropriate where existing code is to be modified
- Research shows that experts judgement in practice tends to be based on analogy

Estimating by analogy

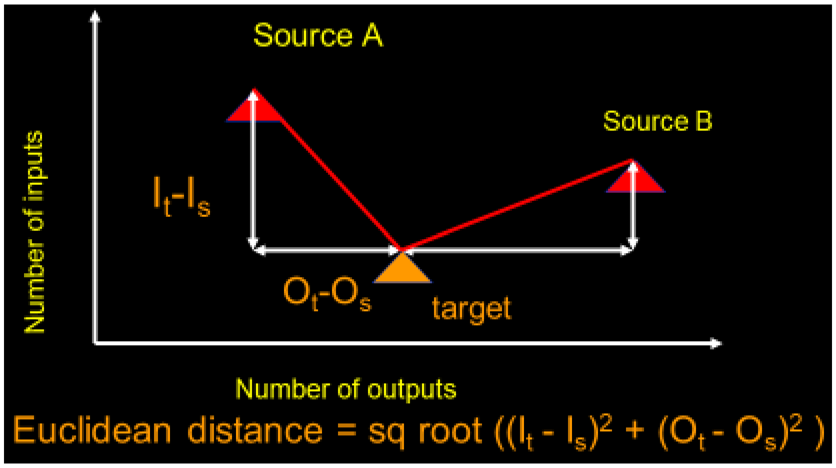


The source cases, in this situation, are completed projects. For each of details of the factors that would have a bearing on effort are recorded. These might include lines of code, function points (or elements of the FP counts such as the number of inputs, outputs etc), number of team members etc etc. For the values for the new project are used to find one or more instances from the past projects than match the current one. The actual effort from the past project becomes the basis of the estimate for the new project.

Stages: identify

- Significant features of the current project
- previous project(s) with similar features
- differences between the current and previous projects
- possible reasons for error (risk)
- measures to reduce uncertainty

Machine assistance for source selection (ANGEL)



Parametric models

We are now looking more closely at four parametric models:

1. Albrecht/IFPUG function points
2. Symons/Mark II function points
3. COSMIC function points
4. COCOMO81 and COCOMO II

Recall that function points model system size, while COCOMO focuses on productivity factors.

Albrecht/IFPUG function points

- Albrecht worked at IBM and needed a way of measuring the relative productivity of different programming languages.
- Needed some way of measuring the size of an application without counting lines of code.
- Identified five types of component or functionality in an information system
- Counted occurrences of each type of functionality in order to get an indication of the size of an information system

Different programming languages have different degrees of 'power' which relate to the ratio between the length of programming commands and the amount of processing they create. This is analogous to trying to assess the productivity of bricklayers where different bricklayers work with bricks of different sizes. One way of dealing with this problem is to say that what is important is the size of the wall being built not the number of bricks it contains. FPs are way of measuring the amount of functionality in an application without counting the lines of code in the application.

Five function types

1. **Logical interface file (LIF) types** – equates roughly to a data store in systems analysis terms. Created and accessed by the target system

2. **External interface file types (EIF)** – where data is retrieved from a data store which is actually maintained by a different application.

3. **External input (EI) types** – input transactions which update internal computer files

4. **External output (EO) types** – transactions which extract and display data from internal computer files. Generally involves creating reports.

5. **External inquiry (EQ) types** – user initiated transactions which provide information but do not update computer files. Normally the user inputs some data that guides the system to the information the user needs.

Albrecht complexity multipliers

External user types	Low complexity	Medium complexity	High complexity
EI	**3**	**4**	**6**
EO	**4**	**5**	**7**
EQ	**3**	**4**	**6**
LIF	**7**	**10**	**15**
EIF	**5**	**7**	**10**

The complexity of each instance of each 'user type' is assessed and a rating applied. Originally this assessment was largely intuitive, but later versions, developed by IFPUG (the International FP User Group) have rules governing how complexity is rated.

Examples

Payroll application has:

1. Transaction to input, amend and delete employee details – an EI that is rated of medium complexity

2. A transaction that calculates pay details from timesheet data that is input – an EI of high complexity
3. A transaction of medium complexity that prints out pay-to-date details for each employee – EO
4. A file of payroll details for each employee – assessed as of medium complexity LIF
5. A personnel file maintained by another system is accessed for name and address details – a simple EIF

What would be the FP counts for these?

FP counts

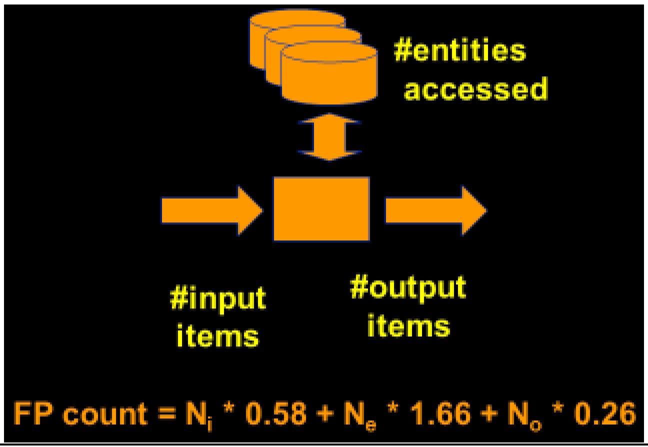
1. Medium EI	4 FPs
2. High complexity EI	6 FPs
3. Medium complexity EO	5 FPs
4. Medium complexity LIF	10 FPs
5. Simple EIF	5 FPs
Total	**30 FPs**

If previous projects delivered 5 FPs a day, implementing the above should take 30/5 = 6 days

Function points Mark II

- Developed by Charles R. Symons
- 'Software sizing and estimating - Mk II FPA', Wiley & Sons, 1991.
- Builds on work by Albrecht
- Work originally for CCTA:
    - should be compatible with SSADM; mainly used in UK
- has developed in parallel to IFPUG FPs
- A simpler method

Mark II FPs is a version of function points developed in the UK and is only used by a minority of FP specialists. The US-based IFPUG method (developed from the original Albrecht approach) is more widely used. I use the Mark II version because it has simpler rules and thus provides an easier *introduction to the principles of FPs. Mark II FPs are explained in more detail in Section 5.11. If you are really keen on teaching the IFPUG approach then look at Section 5.10. The IFPUG rules are really quite tricky in places and for the full rules it is best to consult IFPUG documentation.*



- For each transaction, count
  - data items input (N_i)
  - data items output (N_o)
  - entity types accessed (N_e)

For each transaction (cf use case) count the number of input types (not occurrences e.g. where a table of payments is input on a screen so the account number is repeated a number of times), the number of output types, and the number of entities accessed. Multiply by the weightings shown and sum. This produces an FP count for the transaction which will not be very useful. Sum the counts for all the transactions in an application and the resulting index value is a reasonable indicator of the amount of processing carried out. The number can be used as a measure of size rather than lines of code. See calculations of productivity etc discussed earlier.

There is an example calculation in Section 5.9 (Example 5.3) and Exercise 5.9 should give a little practice in applying the method.

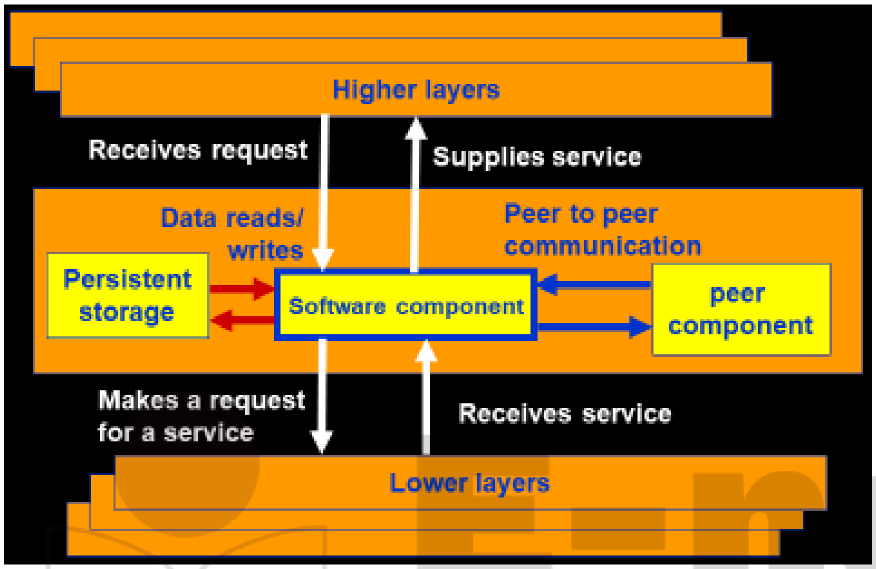
Function points for embedded systems

- Mark II function points, IFPUG function points were designed for information systems environments
- COSMIC FPs attempt to extend concept to embedded systems
- Embedded software seen as being in a particular 'layer' in the system
- Communicates with other layers and also other components at same level

- *Attempts have been made to extend IFPUG FPs to real-time and embedded systems, but this has not been very convincing (IMHO).*

- *Embedded software component is seen as at a particular level in the system. It receives calls for services from a higher layer and requests services from lower layers. It will receive responses from lower levels and will send responses to higher levels.*

Layered software



Each arrow represents an enter or exit if in black, or a read/write if in red.

COSMIC FPs

The following are counted:

- Entries: movement of data into software component from a higher layer or a peer component
- Exits: movements of data out
- Reads: data movement from persistent storage
- Writes: data movement to persistent storage

Each counts as 1 'COSMIC functional size unit' (Cfsu)

Exercise 5.10 gives some practice in applying the technique.

COCOMO81

- Based on industry productivity standards - database is constantly updated
- Allows an organization to benchmark its software development productivity
- **Basic model**

  effort = c x size^k

- C and k depend on the type of system: organic, semi-detached, embedded
- Size is measured in 'kloc' ie. Thousands of lines of code

COCOMO81 is the original version of the model which has subsequently been developed into COCOMO II some details of which are discussed in Section 5.13. For full details read Barry Boehm et al. Software estimation with COCOMO II Prentice-Hall 2002.

The COCOMO constants

System type	**c**	**k**
Organic (broadly, information systems)	**2.4**	**1.05**
Semi-detached	**3.0**	**1.12**
Embedded (broadly, real-time)	**3.6**	**1.20**

k exponentiation – 'to the power of…' adds disproportionately more effort to the larger projects takes account of bigger management overheads

An interesting question is what a 'semi-detached' system is exactly. To my mind, a project that combines elements of both real-time and information systems (i.e. has a substantial database) ought to be even more difficult than an embedded system.

Another point is that COCOMO was based on data from very large projects. There are data from smaller projects that suggest larger projects tend to be more productive because of economies of scale. At some point the diseconomies of scale caused by the additional management and communication overheads then start to make themselves felt.

Development effort multipliers (dem)

According to COCOMO, the major productivity drivers include:

Product attributes: required reliability, database size, product complexity

Computer attributes: execution time constraints, storage constraints, virtual machine (VM) volatility

Personnel attributes: analyst capability, application experience, VM experience, programming language experience

Project attributes: modern programming practices, software tools, schedule constraints

Virtual machine volatility is where the operating system that will run your software is subject to change. This could particularly be the case with embedded control software in an industrial environment.

Schedule constraints refers to situations where extra resources are deployed to meet a tight deadline. If two developers can complete a task in three months, it does not follow that six developers could complete the job in one month. There would be additional effort needed to divide up the work and co-ordinate effort and so on.

Using COCOMO development effort multipliers (dem)

An example: for analyst capability:

- Assess capability as very low, low, nominal, *high* or very high
- Extract multiplier:

very low	1.46
low	1.19
nominal	1.00
high	*0.80*
very high	0.71

- Adjust nominal estimate e.g. 32.6 x 0.80 = 26.8 staff months

As Time Passed… COCOMO 81 Showed Limitations…

- COCOMO 81 was developed with the assumption:
  - Waterfall process would be used and that all software would be developed from scratch.
- Since its formulation, there have been many changes in software engineering practices:
  - Made it difficult to use COCOMO meaningfully.

Major Changes in Program Development Practices

- Software reuse
- Application generation of programs
- Object oriented approaches
- Need for rapid development
- Agile models

COCOMO II

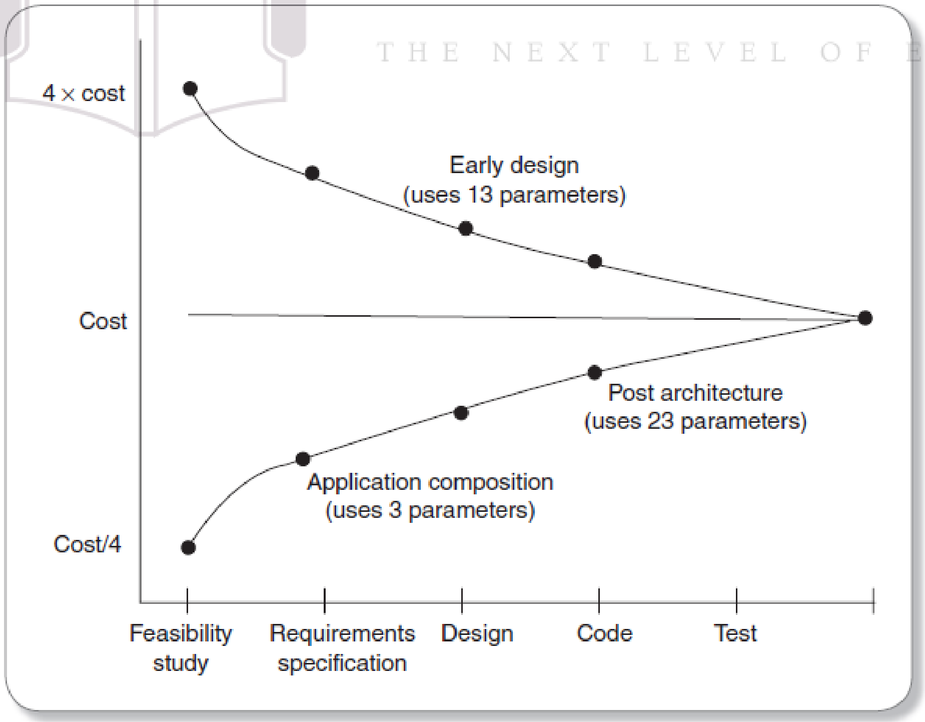


FIGURE 5.4 Accuracy of different COCOMO II estimations

COCOMO II Models

- COCOMO 2 incorporates a range of sub-models:
- Produces increasingly accurate estimates.
- The 4 sub-models in COCOMO 2 are:
  - Application composition model. Used when software is composed from existing parts.
  - Early design model. Used when requirements are available but design has not yet started.
  - Reuse model. Used to compute the effort of integrating reusable components.
  - Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

An updated version of COCOMO:

- There are different COCOMO II models for estimating at the 'early design' stage and the 'post architecture' stage when the final system is implemented. We'll look specifically at the first.
- The core model is:

$pm = A(size)^{(sf)} \times (em_1) \times (em_2) \times (em_3)\ldots.$

where **pm** = person months, **A** is 2.94, **size** is number of thousands of lines of code, **sf** is the scale factor, and **em_i** is an effort multiplier

A could possibly change as more productivity data is collected, but the value of 2.94 remains unchanged since 2000.

Section 5.13.

COCOMO II Scale factor

Based on five factors which appear to be particularly sensitive to system size

1. Precedentedness (PREC). Degree to which there are past examples that can be consulted
2. Development flexibility (FLEX). Degree of flexibility that exists when implementing the project

3. Architecture/risk resolution (RESL). Degree of uncertainty about requirements
4. Team cohesion (TEAM).
5. Process maturity (PMAT) could be assessed by CMMI – see Section 13.10

Exercise 5.11 provides an exercise about the calculation of the COCOMO II scale factor.

COCOMO II Scale factor values

Driver	Very low	Low	Nom-inal	High	Very high	Extra high
PREC	6.20	4.96	3.72	2.48	1.24	0.00
FLEX	5.07	4.05	3.04	2.03	1.01	0.00
RESL	7.07	5.65	4.24	2.83	1.41	0.00
TEAM	5.48	4.38	3.29	2.19	1.10	0.00
PMAT	7.80	6.24	4.68	3.12	1.56	0.00

Example of scale factor

- A software development team is developing an application which is very similar to previous ones it has developed.
- A very precise software engineering document lays down very strict requirements. PREC is very high (score 1.24).
- FLEX is very low (score 5.07).
- The good news is that these tight requirements are unlikely to change (RESL is high with a score 2.83).
- The team is tightly knit (TEAM has high score of 2.19), but processes are informal (so PMAT is low and scores 6.24)

Scale factor calculation

The formula for sf is

sf = B + 0.01 × Σ scale factor values

i.e. sf = 0.91 + 0.01

× (1.24 + 5.07 + 2.83 + 2.19 + 6.24)

= 1.0857

If system contained 10 kloc then estimate would be 2.94 x $10^{1.0857}$ = 35.8 person months

Using exponentiation ('to the power of') adds disproportionately more to the estimates for larger applications

Effort multipliers

As well as the scale factor effort multipliers are also assessed:

RCPX	Product reliability and complexity
RUSE	Reuse required
PDIF	Platform difficulty
PERS	Personnel capability
FCIL	Facilities available
SCED	Schedule pressure

Effort multipliers

	Extra low	Very low	Low	Nom-inal	High	Very high	Extra high
RCPX	**0.49**	**0.60**	**0.83**	**1.00**	**1.33**	**1.91**	**2.72**
RUSE			**0.95**	**1.00**	**1.07**	**1.15**	**1.24**

PDIF			**0.87**	**1.00**	**1.29**	**1.81**	**2.61**
PERS	**2.12**	**1.62**	**1.26**	**1.00**	**0.83**	**0.63**	**0.50**
PREX	**1.59**	**1.33**	**1.12**	**1.00**	**0.87**	**0.74**	**0.62**
FCIL	**1.43**	**1.30**	**1.10**	**1.00**	**0.87**	**0.73**	**0.62**
SCED		**1.43**	**1.14**	**1.00**	**1.00**	**1.00**	

Example

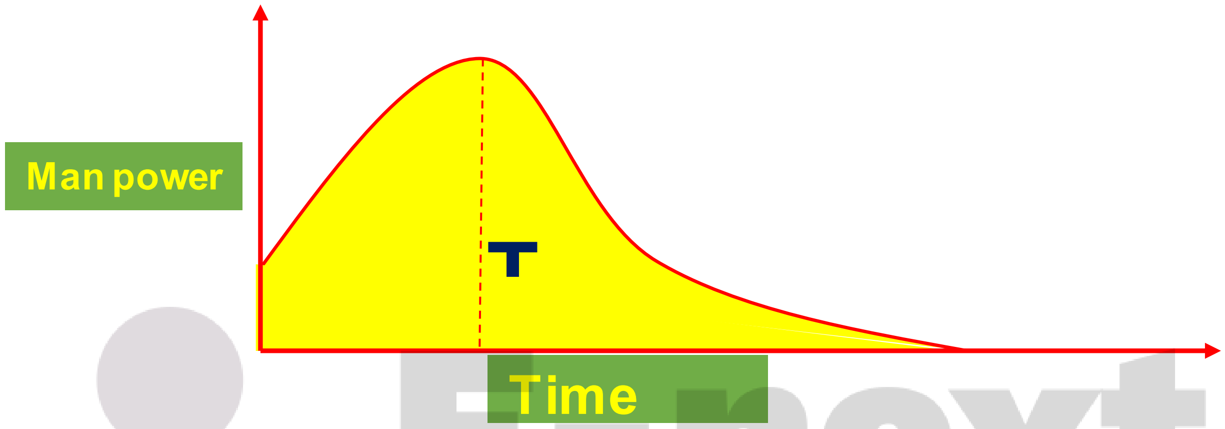
- Say that a new project is similar in most characteristics to those that an organization has been dealing for some time
- **except**
  - the software to be produced is exceptionally complex and will be used in a safety critical system.
  - The software will interface with a new operating system that is currently in beta status.
  - To deal with this the team allocated to the job are regarded as exceptionally good, but do not have a lot of experience on this type of software.

RCPX	very high	1.91
PDIF	very high	1.81
PERS	extra high	0.50
PREX	nominal	1.00

- All other factors are nominal
- Say estimate is 35.8 person months
- With effort multipliers this becomes 35.8 x 1.91 x 1.81 x 0.5 = 61.9 person months

Staffing

- Norden was one of the first to investigate staffing pattern:
    - Considered general research and development (R&D) type of projects.
- Norden concluded:
    - Staffing pattern for any R&D project can be approximated by the Rayleigh distribution curve



Putnam's Work

- Putnam adapted the Rayleigh-Norden curve:
    - Related the number of delivered lines of code to the effort and the time required to develop the product.
    - Studied the effect of schedule compression:

$$pm_{new} = pm_{org} \times \left(\frac{td_{org}}{td_{new}} \right)^4$$

Example

- If the estimated development time using COCOMO formulas is 1 year:
    - Then to develop the product in 6 months, the total effort required (and hence the project cost) increases 16 times.

Boehm's Result

- There is a limit beyond which a software project cannot reduce its schedule by buying any more personnel or equipment.
  - This limit occurs roughly at 75% of the nominal time estimate for small and medium sized projects

Capers Jones' Estimating Rules of Thumb

- Empirical rules:
  - Formulated based on observations
  - No scientific basis
- Because of their simplicity,:
  - These rules are handy to use for making off-hand estimates.
  - Give an insight into many aspects of a project for which no formal methodologies exist yet.

Capers Jones' Rules

- *Rule 1: SLOC-function point equivalence:*
  - One function point = 125 SLOC for C programs.
- Rule 2: Project duration estimation:
  - Function points raised to the power 0.4 predicts the approximate development time in calendar months.
- Rule 3: Rate of requirements creep:
  - User requirements creep in at an average rate of 2% per month from the design through coding phases.
- Rule 4: Defect removal efficiency:
  - Each software review, inspection, or test step will find and remove 30% of the bugs that are present.

- Rule 5: Project manpower estimation:
    - The size of the software (in function points) divided by 150 predicts the approximate number of personnel required for developing the application.
- Rule 6: Number of personnel for maintenance
    - *Function points divided by 500 predicts the approximate number of personnel required for regular maintenance activities.*
- Rule 7: Software development effort estimation:
    - The approximate number of staff months of effort required to develop a software is given by the software development time multiplied with the number of personnel required.

Some conclusions: how to review estimates

Ask the following questions about an estimate

- What are the task size drivers?
- What productivity rates have been used?
- Is there an example of a previous project of about the same size?
- Are there examples of where the productivity rates used have actually been found?