The Economics of the Design Process

Hubert Matthews, Oxyware Ltd
hubert@oxyware.com
http://www.oxyware.com
ACCU Spring Conference 2004

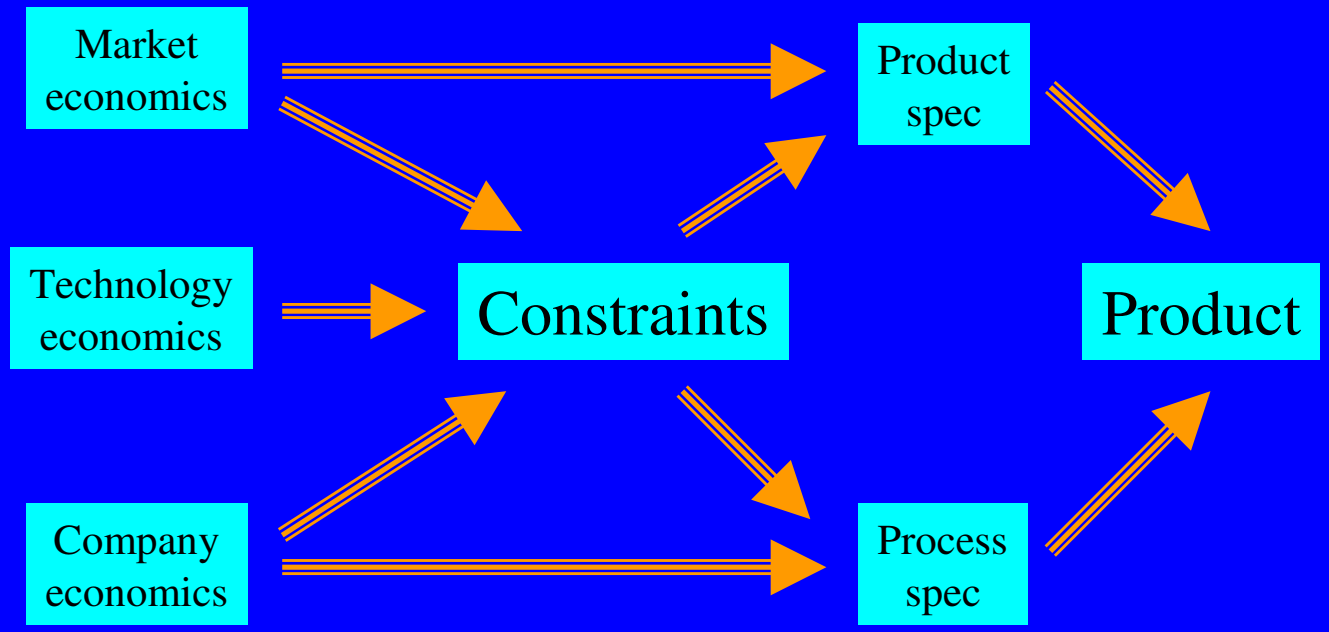
Introduction

- How do
  - the economics of the market,
  - of technology
  - and of our company
- influence the
  - design of our product
  - and the process we use to create it?

Economic Analysis

- Sensitivity of project "success" to cost of:
  - Delay, unit price, development cost, performance/quality
- Basis for project-level tradeoffs
  - Everyone knows what "good enough means"
  - Local tradeoffs line up with global goal
- But there are always constraints (limits)
  - Commercial, technology, vendor
  - Time, cost, quality, bandwidth, CPU, disk, usability, skills, access to hardware, etc….

Constraints are key



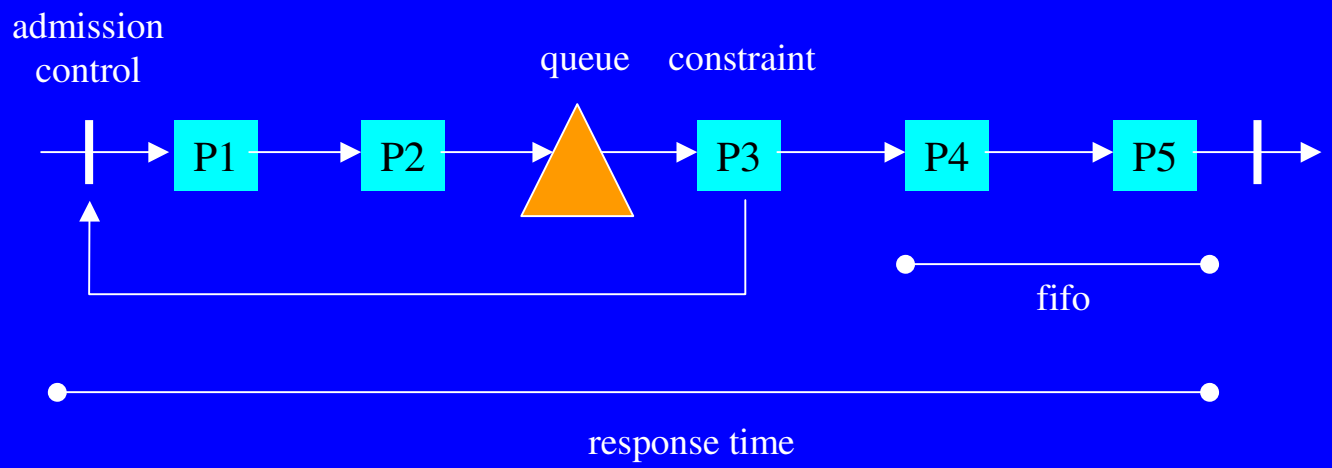
Dealing with constraints

- Goldratt: *Theory of Constraints* ("The Goal")
- Anderson: *Agile Management*
- Identify, Exploit, Subordinate, Elevate, Iterate

1) Find the limiting factor, the bottleneck
2) Use it to the maximum
3) Structure the system to maximise use of the constraint
4) Add capacity to the constraint, if possible
5) Find the new bottleneck

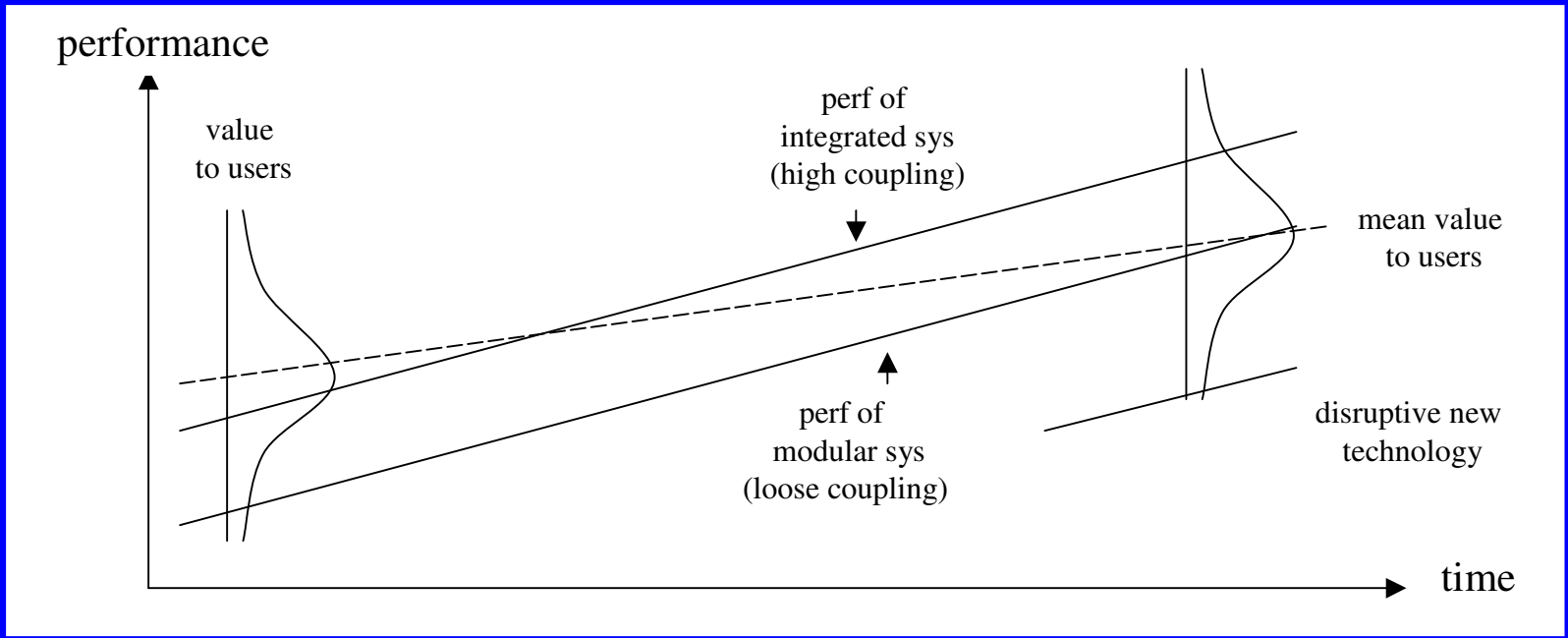
CPU – threading/async, cost – budgets, network latency – coarse-grained calls, access to hardware – simulator, skills – scheduling

Constraints and processes



- Little's Law: response time = (work in process)/(completion rate)
  - Responsive if WIP is low, i.e. small queues
  - Admission control used to keep internal queues small
    - Airplanes, tube stations, not roads!
- Queues - threads, disk I/O, reqts backlog in Scrum/XP
- Sorted queue – triage increases speed of throughput of value

Disruption and maturity



- Moving up-market captures higher margins, but lower volumes
  - Continued upward pressure on performance (current constraint)
- We can afford modular systems when performance is good enough
- New disruptive slower technology enters when other constraints are lifted
  - Cost, usability, time to market, maintainability (Java v. native code, GUI v. command line, etc)

Modularity and standard i/fs

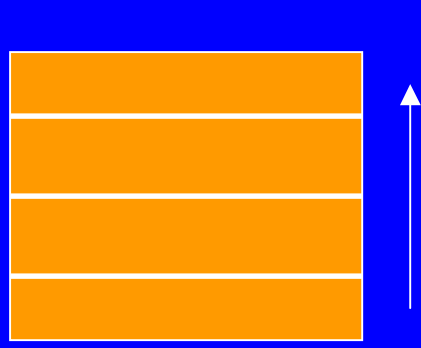
- When performance is not a constraint, modularity is feasible
- This leads to interfaces => standards
- This leads to commoditisation
- Differentiating a product that implements a standard i/f is to make it faster, cheaper, etc
  - Speed and cost need integrated architectures
  - First-tier suppliers profit, not those at the i/f
- Constraints change with time

Changing constraints - example

- Back-end software for producing diaries from scripts

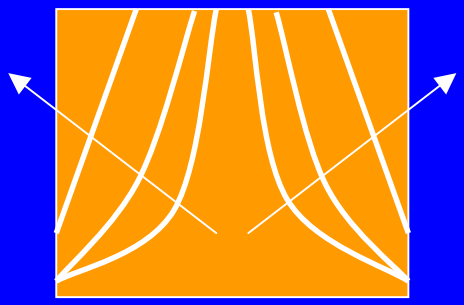
1) Initial development	Low cost, client	Budget
2) Maintenance	Turnaround, me	Time
3) Performance problem	Speed, technical	Performance
4) Marketing	Sales, market	Ease of use

Constraints and creation order



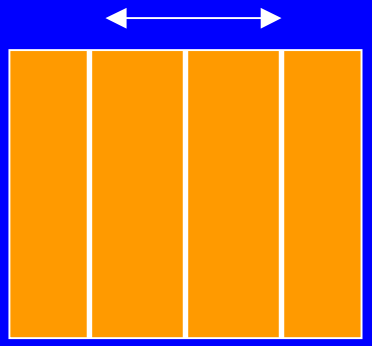
bottom-up
good for programmers

process cost focus



unified process
good for Rational ;-)

balancing cost and delay



value-stream based
good for customers

process delay focus

- Creation order driven by economic analysis

Value of variation

- Software != construction or manufacturing
- There is economic value in producing identical wheels
- There is no economic value in reinventing the wheel
- In manufacturing, variation is usually unwanted (unplanned)
- In design activities such as software, variation is what we're selling (planned)

Types of variation

- Why – planned or unplanned
- When – run-time or design-time
  - product or process
  - early or late binding
- What – function or quality or quantity
- Where – choice points and interfaces
- Who – customer or company
  - external or internal
- How – mechanism

Variation and differentiation

- Variation is a business differentiator (USP)
- Automation is valuable where variation isn't
  - Infrastructure, code generation, checking, testing
- Human judgement is a form of variation
  - We can't automate design (planned)
- Differentiators don't stay that way forever
  - More things become automated over time
  - c.f. model-driven architecture
  - (back to the disruption curve)
- Differentiation needed for growth/revenue
  - Otherwise just do it, cut costs

Handling variation

- Redundancy – buffering
  - Queuing, headroom, “slop”, “padding”
  - Loosely coupled design
- Feed forward – predictive
  - Waterfall, open loop
  - Needs a good predictor, leading metrics
- Feed back – reactive
  - Iterative, closed loop, agile, lagging metrics
  - Can have stability problems
  - Beware partial feedback

Variation and performance

- Lack of variation => redundancy
- Redundancy => “compression”
  - Tight coupling to form of variation

- 3rd normal form exploits redundancy
  - Inheritance similarly
- Caching, HTTP sessions, CVS deltas

Levelling variation

- Where do you “level” variation in your system?
- Do you let variation in or not?
- External
  - Scripting, configuration
  - Reservation system
  - Queue at the boundary (admission control)
- Internal
  - Internal queues
  - Assimilate external use cases

Variability of reqts alters boundaries

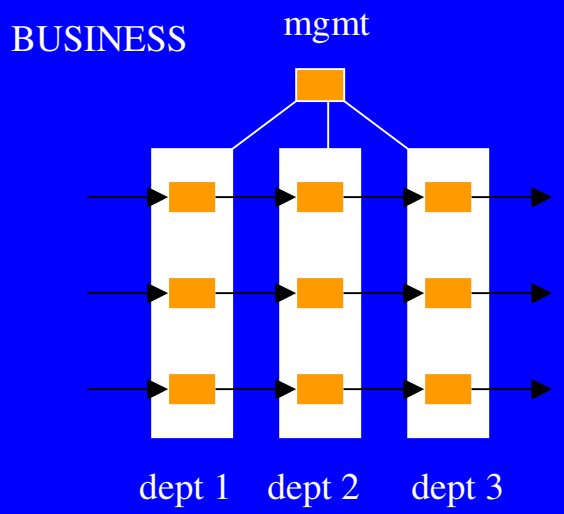
- If reqts are constant
  - Adapt core, centripetal force
- If reqts change often
  - Push them out, centrifugal force
  - Scriptable interface, plug-ins
- Unit of survival
  - Not system
  - But system + environment

Evaluating process steps

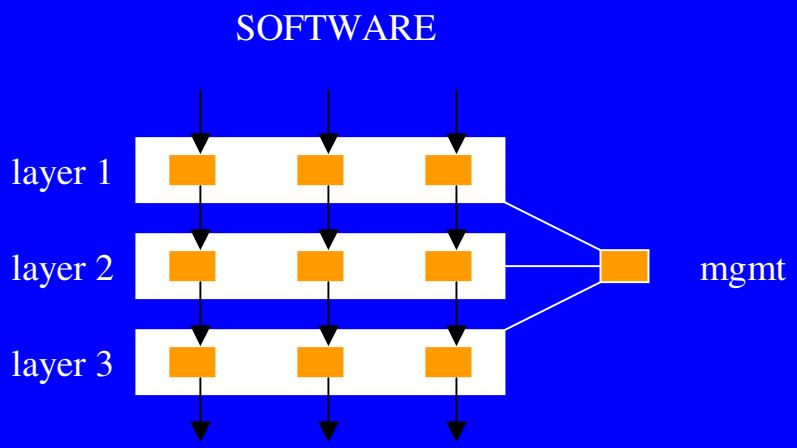
- Valuable
  - lean, usability, value streams, triage, response time
- Capable
  - 6 sigma, feedback-driven, unplanned variation
- Available
  - uptime maintenance, failover
- Adequate
  - capacity, constraints, performance
- Flexible
  - setup time, planned variation

- Both run-time and design-time processes

Internal partitioning



value streams cross depts
local efficiency > global response



value streams cross layers
local modularity > global response

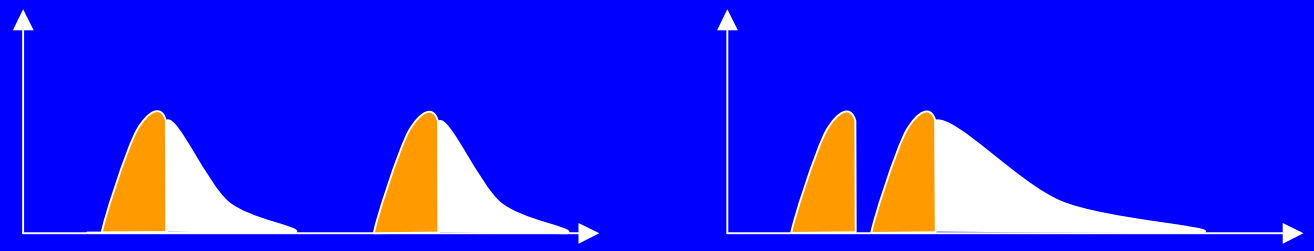
- Use cases as aspects (Jacobsen) – cross cutting

Response times

- If creation time < required response time
  Build to order
  Else
  What?
- Queue + setup + calc + wait time
  - Batches (reduce setup)
  - Caching (historical redundancy in requests)
  - Precalculate (feed forward prediction)
  - Add capacity (buffering)
  - Feedback (admission control, queue length)

Partitioning and budgets

- (Or “Why trains are usually late”)
- Prevent downside propagating
  - Risk containment
- But also prevents upside propagating



- Separate => variations add, no cancellation
- Aggregated => RMS average of variations

Task scheduling

- Student's syndrome
  - Don't start until you've used up all the slack
- No incentive to start early (if you can)
- Time to goldplate and tinker
- Knock-on effect of lateness
- Goldratt's critical chain
  - Aggregate slack as project buffer
  - Monitor buffer usage
  - Buffer of time is equivalent to queue of inventory
  - Protects constraint, i.e. project end date

Summary

- Economic analysis
- Disruption curve
- Integrated v. modular architectures
- Constraints
- Planned v. unplanned variation
- Means of handling that variation
- Reactive v. predictive
- Local v. global optimisation
- How all of these will change over time