Designing for recovery

New challenges for large-scale, complex IT systems

Prof. Ian Sommerville
School of Computer Science
St Andrews University
Scotland

St Andrews

- Small Scottish town, on the north-east coast of the UK
- Home of golf
- Scotland's oldest university (founded in 1413)
- Small university focusing on research and teaching excellence

A question to the audience

- A system is designed to maintain the value of some integer variable (say B), and to provide information about B to users.
- The value of this variable [in the world] is X, with the value of X changing over time.
- The system specification states that the value of B should be X
- Sometimes the system reports to users (correctly) that B = X; sometimes the system reports to users that B = Y, where Y < X
- In circumstances where the system reports that B = Y (i.e. it provides an incorrect value), is this a failure?

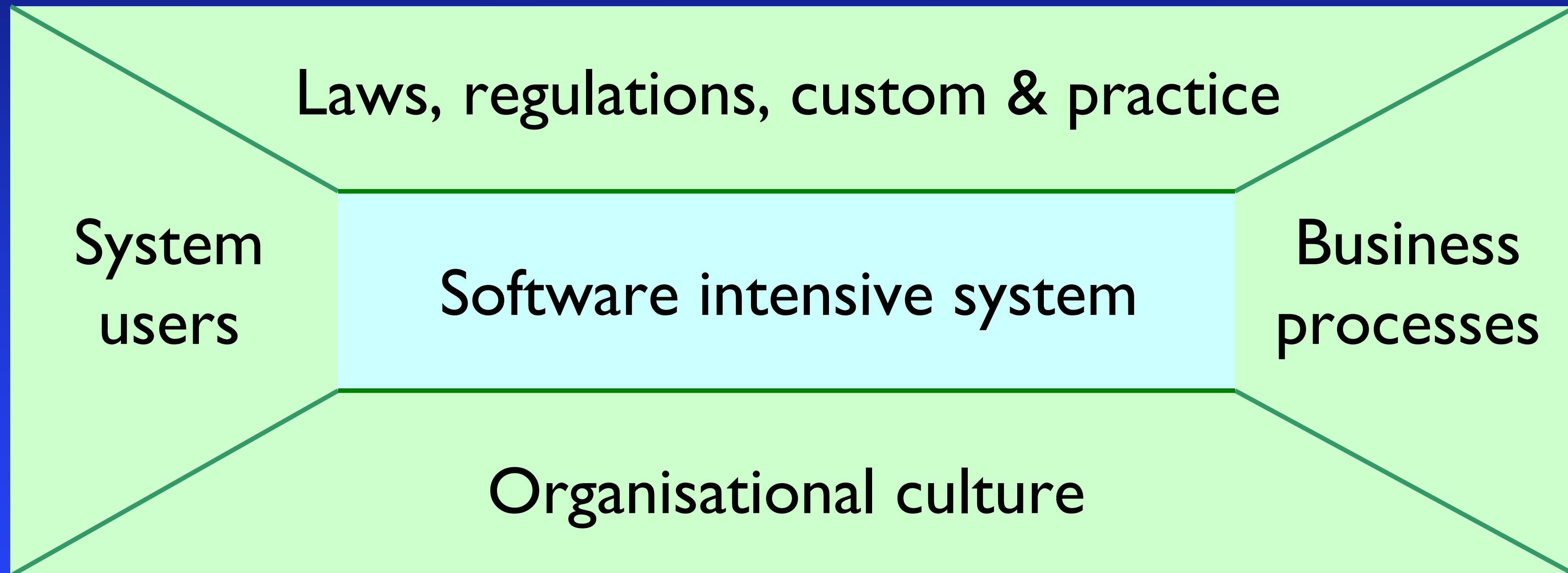
Complex IT systems

- Organisational systems that support different functions within an organisation
- Can usually be considered as systems of systems, ie different parts are systems in their own right
- Usually distributed and normally constructed by integrating existing systems/components/services
- Not subject to limitations derived from the laws of physics (so, no natural constraints on their size)
- Data intensive, with very long lifetime data
- An integral part of wider socio-technical systems

Characteristics of complex IT systems

- Operational independence of the system elements
- Managerial independence of the system elements
- Multiple stakeholder viewpoints
- Evolutionary development
- Emergent behaviour
- Geographic distribution

Socio-technical systems



Reductionism

- Reductionism
    - “an approach to understanding the nature of complex things by reducing them to the interactions of their parts, or to simpler or more fundamental things”.
- Reductionism underpins most engineering, including software engineering
- Reductionism has problems with scale.
    - When things get too big, then reductionist approaches become intellectually unmanageable because of the complexity of the interactions between the parts of the whole

Software engineering

- Developments in software engineering have largely adopted a reductionist perspective:
  - Design methodologies
  - Formal methods
  - Agile approaches
  - Software architecture
  - Model-driven engineering
- Reductionist approaches to software engineering have been successful in allowing us to construct larger software systems
- More effective reductionist approaches allow us to deal with increasingly complicated systems.

Reductionist assumptions

- Control
  - Reductionist approaches assume that we have control over the organisation of the system. It is then possible to decompose the system into parts that can themselves be engineered using reductionist approaches
- A rational world
  - Reductionist approaches assume that rationality will be the principal influence in decision making
- Definable problems
  - Reductionist approaches assume that the problem can be defined and the system boundaries established

Complex and complicated systems

- Reductionist approaches are intended to help deal with complicated systems i.e. systems where there are many interactions between components but which can (in principle) be understood and controlled
- However, we are now building complex systems where is is impossible to acquire and maintain a complete understanding of the system and where elements are independently controlled and often have undocumented side-effects

Services = complexity

What is failure?

- From a reductionist perspective, a failure can be considered to be ‘a deviation from a specification’.
- An oracle can examine a specification and observe a system’s behaviour and detect failures.
- Failure is an absolute - the system has either failed or it hasn’t
- Of course, some failures are more serious than others; it is widely accepted that failures with minor consequences are to be expected and tolerated

A question to the audience

- A hospital system is designed to maintain information about available beds for incoming patients and to provide information about the number of beds to the admissions unit.
- It is assumed that the hospital has a number of empty beds and this changes over time. The variable B reflects the number of empty beds known to the system.
- Sometimes the system reports that the number of empty beds is the actual number available; sometimes the system reports that fewer than the actual number are available .
- In circumstances where the system reports that an incorrect number of beds are available, is this a failure?

Bed management system

- The percentage of system users who considered the system's incorrect reporting of the number of available beds to be a failure was 0%.
- Mostly, the number did not matter so long as it was greater than 1. What mattered was whether or not patients could be admitted to the hospital.
- When the hospital was very busy (available beds = 0), then people understood that it was practically impossible for the system to be accurate.
- They used other methods to find out whether or not a bed was available for an incoming patient.

Failure is a judgement

- Specifications are a simplification of reality
- Users don't read and don't care about specifications
- Whether or not system behaviour should be considered to be a failure, depends on the judgement of an observer of that behaviour
- This judgement depends on:
  - The observer's expectations
  - The observer's knowledge and experience
  - The observer's role
  - The observer's context or situation
  - The observer's authority

System failure

- Failures are not just catastrophic events but normal, everyday system behaviour that disrupts normal work and that mean that people have to spend more time on a task than necessary
- A system failure occurs when a direct or indirect user of a system has to carry out extra work, over and above that normally required to carry out some task, in response to some inappropriate system behaviour
- This extra work constitutes the cost of recovery from system failure

Failures are inevitable

- Technical reasons
  - When systems are composed of opaque and uncontrolled components, the behaviour of these components cannot be completely understood
  - Failures often can be considered to be failures in data rather than failures in behaviour
- Socio-technical reasons
  - Changing contexts of use mean that the judgement on what constitutes a failure changes as the effectiveness of the system in supporting work changes
  - Different stakeholders will interpret the same behaviour in different ways because of different interpretations of 'the problem'

Conflict inevitability

- Impossible to establish a set of requirements where stakeholder conflicts are all resolved
- Therefore, successful operation of a system for one set of stakeholders will inevitably mean 'failure' for another set of stakeholders
- Groups of stakeholders in organisations are often in perennial conflict (e.g. managers and clinicians in a hospital). The support delivered by a system depends on the power held at some time by a stakeholder group.

Where are we?

- Large-scale information systems are inevitably complex systems
- Such systems cannot be created using a reductionist approach
- Failures are a judgement and this may change over time
- Failures are inevitable and cannot be engineered out of a system

The way forward

- Systems design has to be seen as part of a wider process of socio-technical systems engineering
- We need to accept that technical system failures will always occur and examine how we can design these systems to allow the broader socio-technical systems to recognise, diagnose and recover from these failures

Software dependability

- A reductionist approach to software dependability takes the view that software failures are a consequence of software faults
- Techniques to improve dependability include
  - Fault avoidance
  - Fault detection
  - Fault tolerance
- These approaches have taken us quite a long way in improving software dependability. However, further progress is unlikely to be achieved by further improvement of these techniques as they rely on a reductionist view of failure.

Failure recovery

- Recognition
  - Recognise that inappropriate behaviour has occurred
- Hypothesis
  - Formulate an explanation for the unexpected behaviour
- Recovery
  - Take steps to compensate for the problem that has arisen

Coping with failure

- Socio-technical systems are remarkably robust because people are good at coping with unexpected situations when things go wrong.
  - We have the unique ability to apply previous experience from different areas to unseen problems.
  - Individuals can take the initiative, adopt responsibilities and, where necessary, break the rules or step outside the normal process of doing things.
  - People can prioritise and focus on the essence of a problem

Recovering from failure

- Local knowledge
  - Who to call; who knows what; where things are
- Process reconfiguration
  - Doing things in a different way from that defined in the 'standard' process
  - Work-arounds, breaking the rules (safe violations)
- Redundancy and diversity
  - Maintaining copies of information in different forms from that maintained in a software system
  - Informal information annotation
  - Using multiple communication channels
- Trust
  - Relying on others to cope

Design for recovery

- The aim of a strategy of design for recovery is to:
  - Ensure that system design decisions do not increase the amount of recovery work required
  - Make system design decisions that make it easier to recover from problems
    - Earlier recognition of problems
    - Visibility to make hypotheses easier to formulate
    - Flexibility to support recovery actions
- Designing for recovery is a holistic approach to system design and not (just) the identification of 'recovery requirements'
- Should support the natural ability of people and organisations to cope with problems

Problems

- Security and recoverability
- Automation hiding
- Process tyranny
- Multi-organisational systems

Security and recoverability

- There is an inherent tension between security and recoverability
- Recoverability
  - Relies on trusting operators of the system not to abuse privileges that they may have been granted to help recover from problems
- Security
  - Relies on mistrusting users and restricting access to information on a ‘need to know’ basis

Automation hiding

- A problem with automation is that information becomes subject to organizational policies that restrict access to that information.
- Even when access is not restricted, we don't have any shared culture in how to organise a large information store
- Generally, authorisation models maintained by the system is based on normal rather than exceptional operation.
- When problems arise and/or when people are unavailable, breaking the rules to solve these problems is made more difficult.

Process tyranny

- Increasingly, there is a notion that ‘standard’ business processes can be defined and embedded in systems that support these processes
- Implicitly or explicitly, the system enforces the use of the ‘standard’ process
- But this assumes three things:
  - The standard process is always appropriate
  - The standard process has anticipated all possible failures
  - The system can be respond in a timely way to process changes

Multi-organisational systems

- Many rules enforced in different ways by different systems.
- No single manager or owner of the system . Who do you call when failures occur?
- Information is distributed - users may not be aware of where information is located, who owns information, etc.
- Processes involve remote actors so process reconfiguration is more difficult
- Restricted information channels (e.g. help unavailable outside normal business hours; no phone numbers published, etc.)
- Lack of trust. Owners of components will blame other components for system failure. Learning is inhibited and trust compromised.

Design guidelines

- Local knowledge
- Process reconfiguration
- Redundancy and diversity
- Trust

Local knowledge

- Local knowledge includes knowledge of who does what, how authority structures can be bypassed, what rules can be broken, etc.
- Impossible to replicate entirely in distributed systems but some steps can be taken
  - Maintain information about the provenance of data
    - Who provided the data, where the data came from, when it was created, edited, etc.
  - Maintain organisational models
    - Who is responsible for what, contact details

Process reconfiguration

- Make workflows explicit rather than embedding them in the software
  - Not just ‘continue’ buttons! Users should know where they are and where they are supposed to go
- Support workflow navigation/interruption/restart
- Design systems with an ‘emergency mode’ where the the system changes from enforcing policies to auditing actions
  - This would allow the rules to be broken but the system would maintain a log of what has been done and why so that subsequent investigations could trace what happened
- Support ‘Help, I’m in trouble!’ as well as ‘Help, I have a problem?’

Redundancy and diversity

- Maintaining a single 'golden copy' of data may be efficient but it may not be effective or desirable
  - Encourage the creation of 'shadow systems' and provide import and export from these systems
- Allow schemas to be extended
  - Schemas for data are rarely designed for problem solving. Always allow informal extension (a free text box) so that annotations, explanations and additional information can be provided
- Maintain organisational models
  - To allow for multi-channel communications when things go wrong

Trust

- Trust is inherent in problem recovery as it involves trusting people to be well-intentioned and to focus on solving the problem rather than on narrower concerns
- As we move away from co-located systems, which allow personal relationships to be created, establishing trust becomes more and more difficult
- There is some research on 'trust models' but it is not clear (to me) how this can be applied to recoverability

Current research

- Our current work is concerned with the development of responsibility models that make responsibilities across different organisations explicit
- These models show who is responsible for what and the resources required to discharge responsibilities
- They provide a basis for maintaining local knowledge about a situation and discovering who to involve when problems have to be solved

Summary

- A reductionist approach to software engineering is no longer viable. on its own, for complex systems engineering
- Improving existing software engineering methods will help but will not deal with the problems of complexity that are inherent in distributed systems of systems
- We must learn to live with normal, everyday failures
- Design for recovery involves designing so that the work required to recover from a failure is minimised
- Recovery strategies include supporting information redundancy and annotation and maintaining organisational models