

---

# Extreme Programming for Critical Systems?

Ian Sommerville

(with thanks to Simon Monk, momote  
Ltd for discussions of the reality of XP)

# Critical Systems

---

- Systems whose failure has serious consequences for both the organisation running the system and the 'outside world'
  - **Business critical systems.** Systems whose failure could threaten the stability of the business
  - **Mission critical systems.** Systems whose failure could result in the failure of some wider activity
  - **Safety critical systems.** Systems whose failure could endanger human life

# Critical Systems Characteristics

---

- Socio-technical, not just technical systems
- Regulation and compliance
- Systemic dependability requirements
- Long-lifetime (Often but not always)

# Socio-technical Systems

---

- The system is not just the software or the software+hardware but also includes users, processes, procedures, rules and regulations, etc.
- Some system 'failures' result from failures of elements outside the technical system or because of mismatches between technical and social systems
- BUT, generally, the social system is essential for recovery from failure

# Regulation and compliance

---

- Critical systems (especially safety critical systems but increasingly other classes of system) are often subject to regulation by an outside regulator
- Unless the regulator is satisfied that the system is trustworthy, it may not be possible to deploy the system
- In other business-critical systems, failure to comply to regulations can lead to very high business costs

# Systemic dependability requirements

---

- Many requirements of critical systems do not relate to individual functionality but are systemic i.e. they apply to the system as a whole
- Systemic requirements are usually negotiated across multiple stakeholders
- Dependability requirements are concerned with the reliability, availability, safety, integrity, confidentiality, etc. of the system
- Dependability requirements generate functional system requirements that are NOT user requirements

# Long-lifetime

---

- Most critical systems have a long-lifetime
  - The cost of building and installing the system is high so has to be amortised over many years
  - The risks of replacing the system are high so there has to be a compelling reason to do so
- Consequently, developers of critical systems often have to work with obsolete hardware and software technologies
- System structures inevitably degrade over time
- Operational processes change and evolve due to business change and external influences
- There may be little or no organisational memory about the development of the system

# Critical Systems Engineering

---

- Classically, the principal example of 'Big Design Up Front'
- Mostly still developed by a waterfall model
  - Requirements
  - Design
  - Programming
  - Testing and validation
- Reviews are a central part of the critical systems engineering process
- Documentation is often mandated (but, equally often, never read)

# Why XP for Critical Systems?

---

- Critical systems are subject to exactly the same pressures as other types of system
  - Faster delivery
  - Higher quality
  - Lower cost
- Waterfall processes are no better for critical systems engineering than for any other type of system
  - But they do allow people to avoid blame

# Delivering dependability

---

- Defining the right system
- Building the system right
- Designing for failure
- Making the system usable
- Keeping the system working

# Defining the 'right' system

---

- Establishing the system boundaries
- Identifying key stakeholders
- Discovering requirements
- Making sense of these requirements

# XP commentary

---

- XP's focus is on the incremental discovery of user requirements
  - As users rarely know what they want, this makes sense.
- Who decides where to draw the system boundaries?
- How is attention paid to systemic requirements which don't have a single owner?
- What about functional requirements that don't come from users?
- How are the interactions between user requirements, dependability requirements and other systemic requirements analysed?
- Who decides on requirements trade-offs?

# Building the system right

---

- Translating the system requirements into code
- Demonstrating that the developed code meets the system requirements
- Ensuring that the developed code does not behave in ways that compromise the system whether or not the appropriate behaviour is defined in the requirements

# XP commentary

---

- We can learn a lot from XP in this area:
  - Common ownership of code
  - Test-first development
  - Pair programming
  - Refactoring and continuous code improvement
- However, there seems to be a lack of consistent advice on how to do architectural design?
- How effective are developer-written tests?

# Architectural Design

---

- Designing the right system architecture is essential if the dependability requirements on a system are to be satisfied
  - For example, to allow for error recovery, distributed redundancy in the architecture may be essential
- To address this, some XP practitioners introduce an 'architecture spike' into the process
- But at what stage is this possible? What requirements are needed before enough information is available? Will developers really be willing to adopt the radically refactoring that may be involved in architectural change?

# Types of testing

---

- Validation testing
  - Testing that the system behaves as intended by its programmer AND its specifier
  - A successful test is one that executes correctly
- Defect testing
  - Testing the system to uncover defects
  - A successful test is one that fails
  - Defect testing is psychologically hard for system developers and users

# Testing - Quality AND Quantity

---

- Focus on test-first development can lead to complacency - the system passed the tests so it must be OK
- Lots of tests do not mean good tests from either a validation or a defect testing perspective
- For critical systems (especially where a regulator is involved), a demonstration of the coverage provided by tests is essential

# Designing for failure

---

- The predominant approach in software systems design is to design optimistically. The design is geared towards normal, routine operation of the system
- However, critical systems designers must think carefully about designing for failure
  - Including flexibility in the system - not optimising around the 'normal' operational process
  - Including redundant code in the system to detect and correct potential errors and to help recovery from these errors

# XP commentary

---

- XP as a process is an essentially optimistic process
  - “Don’t anticipate problems that might never occur”
- This needs to be supplemented with:
  - “unless the costs and risks associated with these problems have significant external effects”
- Users of a system are not necessarily the best people to judge the risks and costs of system failure

# Making the system usable

---

- Not just an issue of user interface design!
- As critical systems are socio-technical systems, they have to fit into the social environment where they are used
- Therefore, there needs to be a match between the facilities of the system and the contingencies of the operating environment
- And, of course, the user interface has to be designed to avoid user errors and allow recovery from these errors

# XP commentary

---

- The involvement of users in the XP process is an important step forward in improving the reliability of UI design
- BUT:
- Individual users can rarely articulate the social influences on system dependability?
- How can information about mistakes and recovery mechanisms be collected?
- Refactoring user interfaces is potentially dangerous - users often perform actions automatically.

# Keeping the system working

---

- The key requirement here is to maintain dependability in the face of system change
  - Ensuring that proposed system changes are compatible with the overall system dependability requirements
  - Ensuring that code changes are properly implemented and tested
  - Ensuring that code changes don't introduce / reveal errors in existing code
  - Ensuring that changes don't lead to operator errors

# XP commentary

---

- The high quality code created through regular refactoring should lead to more maintainable software so fewer introduced errors
- Automated regression testing should help
  - But beware of ordering effects. Functions added to a system don't always execute after existing functions
- However, if insufficient attention has been paid to the architectural design (and the documentation?), then longer-term maintenance issues may result
- Is XP a long-lifetime process? Can it be applied over a 15-20 year period? What happens when the original XP team are no longer available?

# Conclusions

---

- Critical systems engineering needs agile approaches to development
  - Critical systems developers are rightly conservative but recognise that current approaches are too expensive and take too long
- XP includes a set of good practices that have the potential to contribute to critical systems engineering
- For critical systems engineering, some of these need to change and new practices need to be included in the XP process
- Evangelism and hype won't convince people who end up in court if their software kills people or laws are broken

# And finally

---

- As Kent Beck states in the opening sentence of *Extreme Programming Explained*, "The basic problem of software development is risk."
- He is right - XP has opened the debate on the most effective way to address project risks
- A key challenge for the XP community is how to develop and extend the approach to cope with product as well as project risks