Extreme programming (XP) - dead end or a fresh start for software engineering?

Mike Holcombe,
University of Sheffield

Plan

- **Why are lightweight methodologies such as eXtreme Programming emerging?**
- **What is eXtreme Programming, (XP)?**
- **Does it work?**
- **How can it be improved?**

Lightweight methodologies are approaches which reduce the impact of expensive parts of the software engineering life cycle.

Typically this means reducing the complexity of the process and making it less bureaucratic.

Also, trying to make it more human friendly.

Lightweight methodologies focus on building quality software -

rather than on creating vast amounts of documentation that is often:

incomplete,

inconsistent

and unintelligible!

XP tries to address issues where current software engineering fails i.e.:

the dynamic nature of modern business

by the time the design is done it is out of date;

as business needs change the software has to evolve – this is hard to do.

The four basic principles of XP

Communication

Feedback

Simplicity

Courage

The twelve *sacred* tenets of XP (Beck)

1. **Test first programming.**

Before writing any code programmers build a set of tests.

These tests are run – of course they will fail as no code has been written!

Why would one do this?

To get used to testing *continuously* –

At the end of a session, at the end of the day, whenever a small piece of code has been built -

ALL the test sets are run, this means -

all the relevant unit tests,

all the functional tests.

The test sets are the most important resource and are continually enhanced.

The customer helps to supply tests.

Functional tests are derived from the planning game (see below).

The test sets replace the specification.

If any tests fail the code must be fixed.

2. The planning game

The customer provides business stories and estimates are made about the time to build software to implement the stories.

The customer decides which stories provide the most business value.

Programmers implement the chosen stories.

3. Small, frequent releases
Release early and release often.

4. Always use the SIMPLEST design that adds business value.

5. System metaphor.

Programmers define a handful of classes and patterns that shape the core business problem and solution.

6. On-site customer.

Encourages intense face-to-face dialogue.

7. Refactoring.

Restructuring code without changing its functionality.

Used mainly to SIMPLIFY code – make it more understandable, more maintainable.

8. Pair programming.

Two people - One machine.

All code must be written in this way.

This is *continuous* review and gives a much greater understanding of what is being done.

Pairs swap around frequently.

Different pairs form up regularly.

9. Collective code ownership.

ALL the code belongs to ALL the programmers.

Anyone can change anything.

There are house rules for writing and documenting code and for communicating between teams.

10. Coding standards.

Defines rules for shared code ownership and for communication between different team's code.

Consistent class and method naming.

Everyone should use the same coding styles.

11. Continuous integration.

Code is integrated into the system at least a few times every day.

All unit tests must pass prior to integration.

All relevant functional tests must pass afterwards.

12. Forty hour week.

Tired programmers write poor code and make more mistakes.

It is quite hard to stick with ALL these rules - XP requires *discipline*.

Some teams need a "coach" to ensure that they do stick to XP!

There have been successes as well as failures with XP –

More research is needed.

It is demanding

- **For the programmers – they need to develop all round skills**
- **For the clients – they need to give up more time**
- **For managers – they need to trust their teams more**
- **But it raises the profile of testing!**

Does it work?

- **Some comparative empirical evidence that it does.**
- **Different teams using XP and traditional methods building the same systems for the same client.**
- **XP delivered better quality, quicker with less stress.**

XP and industry

- There has been a rapid growth of software houses, especially in the USA adopting XP
- Several UK companies are using it successfully.
- Seems to be popular in financial services and telcomms – both highly dynamic business areas

What are the problems with XP?

The biggest problem is with the functional test sets.

No guidance is given – it all seems *ad hoc*.

The system metaphor development also needs a more structured approach.

And can it work for big projects?

Proper support is needed

- **Support for managing XP projects**
- **Support for XP testing**
- **Support for training and reinforcing the XP principles**
- **Support for code conventions**

The Observatory

- This is a mechanism for empirical research into software methodologies, not just XP.
- Comparative experiments on the complete software development process
- Observations on the ways in which new methods can be adopted into working companies

Software Hut

- 90 2nd year students in teams of 4-5
- 20 teams, 4 external clients.
- Half the teams use XP the rest use "trad"
- Clients evaluate the end product.
- We evaluate the process.
- We collect lots of data: time sheets, plans, test sets, code, designs etc.

This year's clients

- **Small Firms Federation**
- **National Cancer Screening Service**
- **LearnDirect (UfI)**
- **Dental research organisation**

- **Mainly e-commerce/intranet/database applications.**
- **Using MySQL, PHP, JSP, etc.**

XP in Genesys Solutions

- **Introduced XP in 2000.**
- **A working software house. 25-30 part time staff**
- **Both new projects and maintenance projects**
- **XP adopted in 2000**
- **XP popular with most programmers but not all**
- **It's easy to degenerate into bad habits.**
- **Regular reinforcement of the philosophy is needed.**

An XP intranet

- **We have found that the development of this sort of support has helped Genesys to adopt XP better.**
- **Test environments, planning support, resource estimation, code convention templates, test convention templates etc. are all available.**
- **Their use can be monitored.**

More *XP* tasks

- **Collecting data about the process and using it for process improvement**
- **Supplying data for resource estimations in the planning game.**
- **Checking compliance.**

Where XP needs more work

- **Functional testing.**
- **We are using a simple but effective method for deriving these test sets.**
- **It is based on the X-machine total testing method.**

What is total testing?

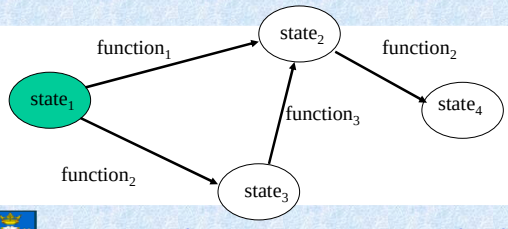
- **Total testing is a technique that finds all functional faults *subject to certain conditions***
- **It is based on computational modelling and has been mathematically proven**
- **It is also very practical and has been used successfully in a number of industrial applications**
- **It requires a functional specification from which the test sets are generated**
- **The specification language used is based on an easy generalisation of finite state machines**

Beyond state machines

- **FSM methods are impractical for most software systems**
- **FSMs are not powerful enough to represent more than the control structure of a system without state explosion problems**
- **They are weak at describing the relationship between control and data which is vital for testing**
- **We introduce a simple concept of *internal memory*.**
- **This is any set of elements that can be used by the machine to model its behaviour**
- **The memory could be the contents of a database or some other internal variables which are needed during the operation of the system**

Generalised machines

- **States are connected by some simple functions**



X-machines 1

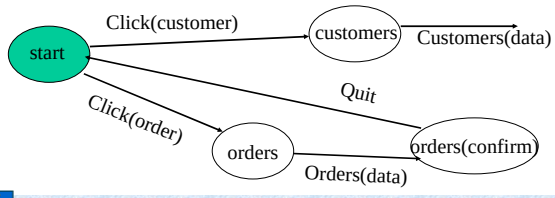
- **We now have a system with:**
  - **states,**
  - **inputs,**
  - **outputs**
  - **and memory**

X-machines 2

- **At the bottom level of the system is a set of basic functions**
- **These functions operate on the basis of being supplied a *pair* of values, the *current input* and the *current state of memory*,**
- **They create an *output* and update the *internal memory* when they operate.**
- **There are no restrictions on the structure or contents of either the memory or these basic functions and this flexibility can be used to abstract and simplify models**

A simple example

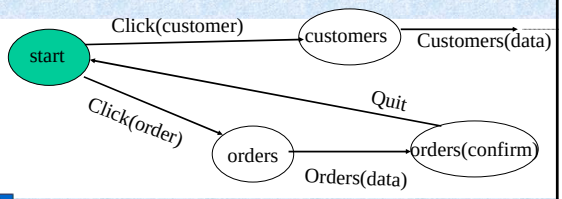
- **The system is then modelled by identifying its states and the basic functions that are the transitions between states.**



X-machines 3

- These simple functions operate as follows:
- Click(order) takes as input a button click from the orders button on the user interface. It ignores the internal memory.
- The result is a state change – here that is a new screen with suitable provision for data entry relating to orders, the output is a new screen, nothing is done to the memory.
- Function: { input, memory; new memory, output }

A simple example (again)

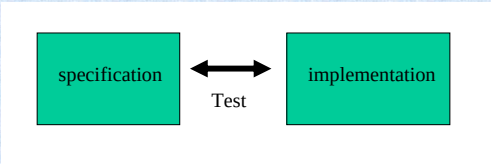


How the model works

- Orders(data) then takes the data entry for all the parameters allowed on the screen and consults the memory to see if there are any conflicts,
- The result is a new screen offering the confirm choice for the data – or the resolution of a conflict if there is one in the data already present
- At this stage the new data is not added to the memory - this happens if the confirm is agreed, carried out by a subsequent function.

Basic testing philosophy

- We have two things that we wish to compare:
- The specification and the implementation – do they ALWAYS behave the same?



Total test fundamentals

- We can only access the implemented system through the system interface
- Thus we can apply inputs and observe the results (outputs)
- On the basis of this we have to determine whether the behaviour is correct
- The test sets generated by the method depend on some assumptions
- If the implementation passes ALL the tests in the test set

THEN IT IS CORRECT – THAT IS IT BEHAVES EXACTLY AND ALWAYS LIKE THE SPECIFICATION

Design for test

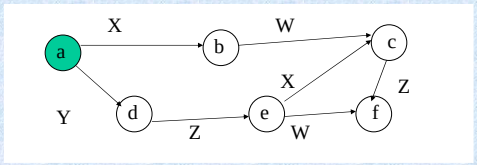
- Assumptions about the specification:
  - It is described as an X-machine
  - The machine is deterministic
  - The basic functions are correct (previously tested/tried and trusted)
  - The specification satisfies the following technical requirements:

Design for test 2

- Controllability – that is, the system can be driven into any state and any function from that state can be exercised
  - Can always be achieved by introducing special test inputs where needed
- Observability – that is, we can determine, purely from external observation which function has been exercised.
  - Can always be achieved by introducing special test outputs where needed

Total test algorithm

- We consider the state space of the specification



Functions: X, Y, Z, W. Initial state:a.

Algorithm

- The test set produced is based on the W method
- We create sequences of functions that visit every state and exercise every possible function from that state – both functions that should be there and those that should not
- We need to identify which state we end up in each time
- This is done by constructing special test sequences that do this
- The complete test set is then a set consisting of sequences of *functions*
- Examples:
  - X :: W :: Z
  - Y :: Z :: X
  - And so on.

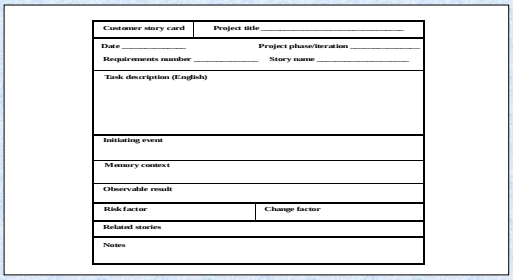
Test set transformation

- We need to transform these sequences into sequences of *inputs* rather than functions
- If the controllability and observability conditions are satisfied then this can be done
- We need to identify the initial state of the internal memory as well
- An algorithm exists to do this
- The test set also requires one further piece of information – an estimate of how many extra states there might be in the implementation - often obtained from code inspection but it could be estimated
- The size of the test set depends on this estimate, more states means more tests

Practical issues

- We need a suitable X-machine specification for this to work
- Many design notations can be converted to this notation
- We need to check that the design for test conditions are satisfied
- We could check that the specification matches the requirements – possibly through model checking
- The test generation process has been automated
- Once testing is finished then we know that the system is correct subject to our assumptions:
  - The basic functions are implemented correctly
  - The estimate of state size is reasonable
  - The design for test conditions hold

From story cards to test sets



Customer story card	Project title
Date	Project phase/iteration
Requirements number	Story name
Task description (English)	
Initiating event	
Memory context	
Observable result	
Risk factor	Change factor
Related stories	
Notes	

Requirements table

From the stories we develop individual requirements in a table format.

story	input	memory	output	new memory	change risk
save	*Save click*	Current d'base	message	updated d'base	1 (low)

Constructing the X-machine

- **From the stories we develop more detailed functional requirement statements**
- **We assemble the X-machine by studying the flow of activity between these functions**
- **Complete the design for test conditions**
- **Generate the test sets automatically.**
- **Refine machine as requirements change**

Generate the test sets

- **Once the design for test and other issues are dealt with.**
- **Test sets generated automatically.**
- **Tests not what the system should do –**
- **Also tests that the system does not do what it shouldn't do.**

Conclusions

- **XP can work well for small to medium projects but testing needs to be done well.**
- **Support is needed – software, management.**
- **It seems popular with programmers.**
- **Not so popular with traditional managers**
- **More empirical research is needed.**

References:

Kent Beck "Extreme programming explained." Addison-Wesley, 1999.

K. Beck & M. Fowler, "Planning extreme programming." Addison-Wesley, 2000.

Ron Jeffries, <http://www.Xprogramming.com>

<http://www.dcs.shef.ac.uk/~wmlh/COM2070XP.html>