Agile + Kernel Programming = ?

Xtreme Challenges require Xtreme Practices?

Mridul Jain

What is the point ?

- Introduction to Agile - mandatory by methodology to turn the dial for good practices to the extreme, always.
- Kernel Programming (KP) – mandatory by necessity to follow extreme practices to make anything work, always.

So, what values can KP draw from Agile?

Kernel Programming in Short

- Coding in kernel space
- Practical limitation of size, memory, limited instruction set.
- Knowledge of hardware required
- Unfettered creativity not allowed - in design, coding etc
- Performance, realtime results matter equally apart from functionality
- Debuggers, performance boosters, maintainers are all kernel programmers to an extent – Kernel knowledge is their common language.
- Surrounded by pseudo kernel programmers` community where everyone considers themselves as Linus or RMS

Agile (from the perspective of KP)

- Agile born in the enterprise world for enterprise technology in user space.
- “What is a customer? We have only heard of users!!”
- “Collective Code ownership + peer review – we already do it in open source!”
- “Sounds same as open source except that we have not heard this word from Eric Raymond.”
- “Oh! We didn’t know that there is something, we are not doing well enough that we now need to depend on some methodology from enterprise?”

Challenges in Kernel Programming

- Consequences of coding in kernel space?
- Problem of bottlenecks at every step.
- Abstract knowledge of concepts and fate of a programmer when he looks at the code!
- Kernel is a single big process
- Where do you start?
- Changes in one place break things elsewhere.
- Unless code follows modularity, things become complex fast & unmanageable.
- Stunts could be costly

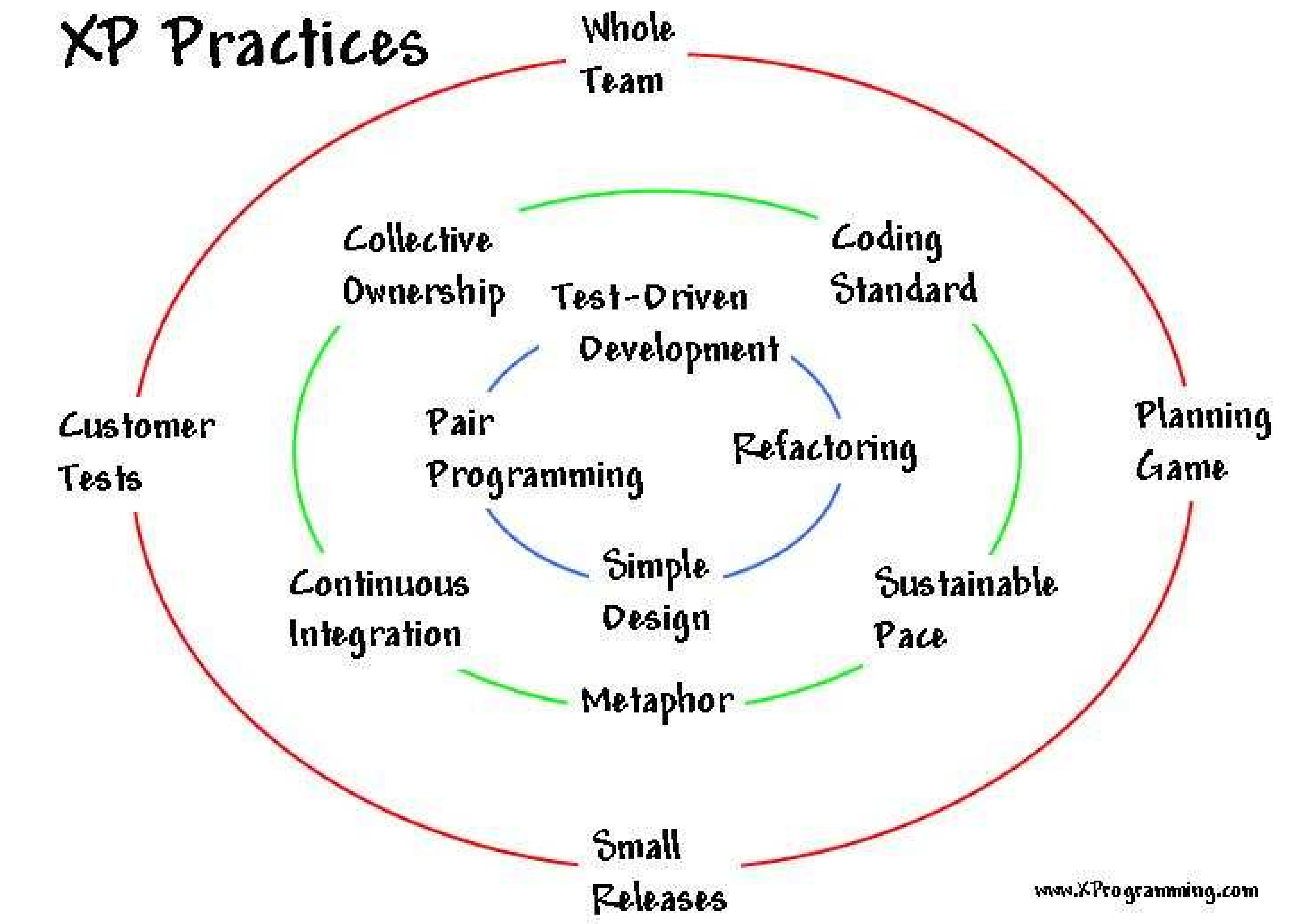
Challenges in following Agile in KP

- Who plays the customer and how much value he gives?
- What will be the customer acceptance test and how will he test them?
- In kernel are requirement cards possible?
- Can requirements be prioritized externally?
- How do estimates and schedules work when most of the steps are bottlenecks? Will the whole project be a spike?
- What happens let's say in case of embedded systems?
- Pair programming is valuable, but how economically feasible is it?
- How does test first approach work? Do we have any test frameworks in kernel space? Should the test be only functional?
- Are frequent builds possible?
- Will pair rotation work?

Agile + KP = hand & glove. Where?

- Iterative nature and design/analysis at every step helps.
- Spikes are not relevant but R&D is(deeper thoughts needed).
- Continuous integration comes naturally
- Courage finds quite relevant meaning naturally.
- Developers are the power. Infact there are only developers and their decision
- Peer discussions is almost second nature.
- Collective Code ownerships seen generally in kernel development.

Finding answers & extending those that exist


XP Practices
Whole Team
Collective Ownership
Coding Standard
Test-Driven Development
Customer Tests
Pair Programming
Refactoring
Planning Game
Continuous Integration
Simple Design
Sustainable Pace
Metaphor
Small Releases
www.XProgramming.com

Test Driven Development

- Need a generic test driver for unit tests
- Linux Test project – Automated testing for kernel
- Contains 2500 test programmes
- Environment for defining new tests, integrating existing benchmarks and analyzing your test results
- LTC has tested more than 50 new kernel versions and found more than 500 defects.
- Includes regression testing on new kernels to ensure they meet the functionality of previous kernels.
- Integration testing then validates component interaction, driven by macro-benchmark workloads.
- Finally, reliability and stress testing validate systemic robustness with extended duration tests (96 hrs to 30 working days).
- Software Testing Automation Framework (STAF/STAX)
- Test coverage visualization tools lets you see code coverage – GCOV/LCOV

LTP GCOV extension - code coverage report

Current view: overview - kernel
Test: gcov.1
Date: 2004-05-17
Code covered: 47.3 %

Instrumented lines: 9195
Executed lines: 4353

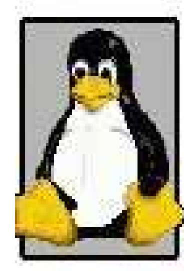
Filename	Coverage		
capability.c		73.0 %	65 / 89 lines
cpu.c		0.0 %	0 / 28 lines
dma.c		0.0 %	0 / 22 lines
exec_domain.c		0.0 %	0 / 99 lines
exit.c		78.6 %	429 / 546 lines
extable.c		35.7 %	5 / 14 lines
fork.c		80.1 %	474 / 592 lines
futex.c		46.1 %	125 / 271 lines

Test Driven Development

- OSDL's Scalable Test Platform
- LTP is one of the tests that OSDL executes
- Web interface also allows to search for historic results.
- Patches placed in the Patch Lifecycle Manager can be analyzed using a variety of test in STP.
- Mode of operation of STP.
- User-Mode Linux is a SAFE, secure-way of running Linux versions and Linux Processes.
- Hack without risking your main Linux setup.
- Less reboots.

Refactoring

- Success of Linux - modularity and clean interfaces
- Great coding - effort of years of constant restructuring from the community. But is it refactoring?
- Need for automated tests systems like LTP/osdl 's STP etc.
- Refactoring can be fruitful by using tools like "Cross-Referencing Linux" http://lxr.linux.no/
- Patch and diff - standard tools by which a restructured code is accepted in the kernel. Need is to send the test scripts which can be added to LTP etc.



Cross-Referencing Linux

Linux/fs/autofs/init.c


```
/* -*- linux-c -*- --------------------------------------------------- *
 *
 * linux/fs/autofs/init.c
 *
 *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
 *
 * This file is part of the Linux kernel and is made available under
 * the terms of the GNU General Public License, version 2, or at your
 * option, any later version, incorporated herein by reference.
 *
 * ----------------------------------------------------------------------- */

#include <linux/module.h>
#include <linux/init.h>
#include "autofs_i.h"

static struct super_block *autofs_get_sb(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data)
{
        return get_sb_nodev(fs_type, flags, data, autofs_fill_super);
}

static struct file_system_type autofs_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "autofs",
```

Simple Design

- Unix philosophy is not a formal design method. More an empirical approach.
- Unix philosophy is bottom-up, not top-down. Pragmatic and grounded in experience.
- Write programs that do one thing and do it well
- Clarity is better than cleverness.
- Separate policy from mechanism; separate interfaces from engines
- Design for simplicity; add complexity only where you must
- Use simple algorithms as well as simple data structures.
- Right choice of data structures, lead to self-evident algorithms.
- Use tools not unskilled help, even if you have to detour to build the tools
- Design and build software, even operating systems, to be tried early, ideally within weeks
- Don't optimize early.

Spikes & KXP

- Notion of spikes does not have much meaning in KXP.
- Lack of knowledge or ignorance cannot be called as a spike in KXP
- Need for a better tool to take care of real R&D issues, things never done before
- R&D and bottlenecks prevent schedule estimation
- Throw R&D/feasibility, outside the usual XP cycle in KXP?

Points to ponder!!

- Sustainable Pace – Lack of information and experience have direct effect on XP.
- Small Releases – Possible but customer here has to be equally experienced developer to test.
- Continuous Integration - Inherent in kernel coding
- Pair Programming – Difficult to find people; but definitely a big help for speedup. More peer programming than pair programming. Falls between books and pair programming – mailing lists, discussions etc.
- Detours and alternatives continuously thought of parallelly and iteratively with analysis & design.
- Definitely a need to do more and write KXP?