SPLC 2009

Adapting to **Change:** *architecture, processes & tools*

A closer look at HP's experience in evolving the OWEN software product line

Jacob Refstrup,
Distinguished Technologist,
Inkjet Systems

hit PRINT

SPLC 2009

What's **Owen**?

- A embedded software product line architecture
- Used in multiple Hewlett-Packard product lines
  - Deskjet, Photosmart, Officejet and Officejet Pro
- First product intro in '98
- Same fundamental architecture in place
  - Evolved architecture/design of subsystems
- Lots of tools & process changes
  - From co-op to full re-use

hit PRINT

hp invent

SPLC 2009

Topics

… a little bit of everything ☺

- Principles
- Owen's architecture
- SCM, build and tools
- Development model
- Variation
- Architectural evolution
- Futures & feature modeling

SPLC 2009

Owen is not perfect or glamorous…

- It's not a perfect architecture / set of processes
  - Has it's share of issues
- You'll find very little earth-shattering in what we do
  - E.g. continuous integration
- It works because of…
  - Hard work
  - Continuous evolution
  - Automation of key processes
  - Balance between structure/flexibility

SPLC 2009

A few…

principles

5 10/28/2009 Hewlett-Packard

hit PRINT

SPLC 2009

Principles

- Should matter who you are not where you are
- Keep it simple
- Best is the enemy of good
- Make it easy to do the right thing; hard to do the wrong thing
- When something is causing pain… do something!
- Don't "bolt" something onto the side – refactor!
- Edge of chaos
- Enforce key rules – otherwise…

6 10/28/2009 Hewlett-Packard

hit PRINT

invent

SPLC 2009

One minute tour of...

architecture

7 10/28/2009 Hewlett-Packard

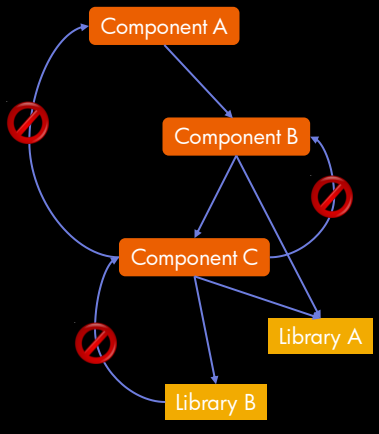
hit PRINT

SPLC 2009

Owen architecture overview

- **Nothing new...**
  - Component based system with client-server topology (including libraries)

- Framework provides key services
- Components
- Libraries
- Shared resources
  - Mostly memory
- Some assets are run-time data driven



8 10/28/2009 Hewlett-Packard

hit PRINT

Quick history of OWEN

- Started as co-op in '97
  - Two geographies; mainly sharing print-engine + framework
  - Over-the-wall tar-ball approach → SCM bridges (low frequency)
  - Two products; < 50 components
- Then…
  - More products, lots of branching & merging (on a product basis)
  - Divergent subsystems
  - Overall leadership/sponsorship fizzled
  - Inconsistencies in build (e.g. version of compiler used)
- There were problems/issues but…
  - They didn't cause enough pain
  - And we shipped plenty of products
- But then… (circa '04)
  - Needing to share more – but incompatible subsystems…
  - Problems got too big
- So…
  - Formed small empowered technical + business teams to address issues
  - Sponsor + leadership identified
  - Lots of improvement projects kicked off
  - Bridged multiple SCM systems close to real-time
- '04/'05
  - Same tools (build, single SCM system, compiler)
- '06 → '07
  - Migrated from multiple branches (one or more per major subsystem) to single development branch
  - Converged on defect tracking tool, requirements tool and document sharing
- Now
  - Five geographies; re-use / sharing everything (directly off trunk)
  - > 800 components; > 20 projects

Sharing **tools**

- Needs to be done
  - It's not sexy
  - Requires investment (people, process, culture, …)
  - And should be done from the beginning…
- Otherwise
  - Lose reproducibility
  - Chaos ensues (the unwanted kind)
- So… make sure you can easily
  - Add tools from vendors
  - Support multiple versions
  - Add your own tools
  - And make them available for everybody everywhere

SPLC 2009

SCM lessons

- The obvious:- one SCM tool
  - Makes merging/branching much easier
- Branching
  - Branching strategy depends on maturity / culture of development organization
  - Cost of branching vs. cost of turmoil
- Ideal tool for SPL?
  - Change-set based systems
  - Need to tailor processes / branching to capability of SCM tool

- Owen's SCM current state
  - **Note: Don't copy unless …**
  - Development on trunk
    - Unless would break build/run-time for extended period
  - Each project has own soft-freeze and hard-freeze branch
    - All changeset marked as defect fixes goes to all current soft-freeze branches
    - Project integrator chooses which changesets goes from SF to HF branch
  - Implies
    - Variability done by build-time / run-time configuration (not SCM configuration)

… how do we make it work?
→ people, development model, variation & continuous integration

SPLC 2009

People & development model

- In the beginning…
  - Small product teams (5-10 developers); "touch" whatever part of source code needed
  - (Coordinated) evolution of subsystems required inter project coordination + lots of merging
- Current situation
  - > 200 engineers working on trunk
  - Can't have everybody stepping on-top of each other
  - Most engineers work within a few subsystem
  - Requires coordination in requirements and execution when spanning subsystems

- Component ownership model
  - Let the engineers know what's expected – accountable for their components.
  - "Owner" doesn't have to do all work
  - A "fixme" process for quick fixes applies to a single project but not suitable for all products (done via build system)
- Architectural implications
  - Evolve architecture such that fewest number of subsystems are involved in new feature development
  - More generally, avoid coupling
- Still evolving…
  - Need to enable more agile development teams
  - Make requirements process more fluid

SPLC 2009

Variation

… how we ship >20 products/year

- The approach
  - Component selection
  - Build flag setting
  - Project header files
  - A BSP-like package unique to each PCA
- Complexity
  - > 800 components
  - > 2000 build flags
    - most dual valued
  - Too many combinations
    - < 100 valid combinations

```
COMPONENTS += comp_a
FEATURE_X = on
COMPONENTS += $(x_COMPONENTS_$(FEATURE_X))
x_COMPONENTS_on += comp_b
```

- A few things we've learned along the way…
  - Naming for the long-term is difficult!
  - Avoid use of project names in source code & build variables
    - "Platform" names can be useful; use in sub-directory names
    - Keep product sub-directories to a minimum
  - Define what are truly top-level build flags
  - Avoid piggy-backing of someone else's build flag
  - Validate build variable settings
  - Keep makefiles DRY
  - Makefile lazy evaluation is hard; but it is really powerful…
- Observations…
  - This ain't good enough
  - Too much of an art-form
  - Need some kind of feature-modeling
    more on that later…

SPLC 2009

Continuous integration

Unwritten rule

"if you never break a build, you aren't working fast enough"

- Joe Bauman, Owen Architect

SPLC 2009

Continuous integration + build system

- Automated builds
  - Whenever new code appears on trunk start builds of all* active projects
  - Lots of emails ☺
  - Expect develop to fix within reasonable time period
    - E.g. immediately, 4 hours, 1 day
- Testing
  - Builds are tested automatically w/small test suite on real HW
  - Some tests executed manually
- Process is ~24/7
  - There's always an accountable person to chase down build/test failures

* Not really all… but close enough

- Build system
  - **Needs to be VERY fast**
  - Makefile driven
  - Linux
  - Fast multi-core machines
- Automated/nightly builds
  - Distributed
- Helper tools
  - Check if a component builds correctly in all active projects

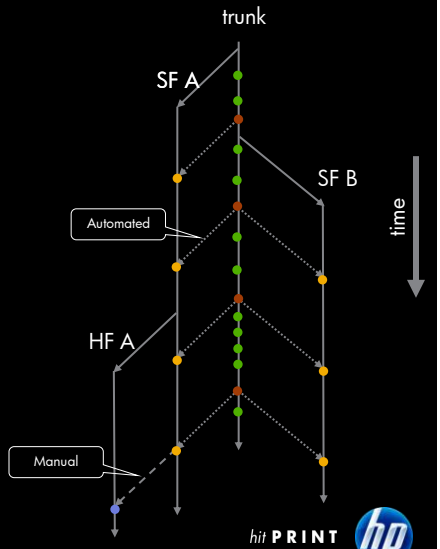
15 10/28/2009 Hewlett-Packard

hit PRINT

SPLC 2009

Stabilizing projects with an **ever evolving trunk**

- Check-in template – choose which projects (branches) need the change
- Defect fixes automatically flows to all active soft-freeze branches
- Automatically flow changesets to picked projects (branches)
- Implies cherry-picking
- Keep branched projects "alive" in trunk
  - >95% of changes can be done in trunk



16 10/28/2009 Hewlett-Packard

hit PRINT

SPLC 2009

Code maintenance

"Whatever 'rock' I turn over I find something…"

Holt Mebane, Owen Architect

SPLC 2009

Spring cleaning…

- What's obsolete - will never be used again?
  - Code fragment, a component, subsystem, a DASIC, a build flag, …
- When something starts feeling wrong – do something; don't put it off.
- Refactor
  - To simplify, remove redundancy, add functionality, …
- Part of normal development process – not just in "spring" ☺
- Write tools to help you & fellow developers
- Some useful tools…
  - Compare build settings before and after making configuration changes
  - Matrix of all build variable settings/project
  - Generated tree-view of makefile inclusion processes
  - Tool to fold/rewrite CPP expressions

SPLC 2009

Architectural

evolution

SPLC 2009

Architectural evolution
A brief summary…

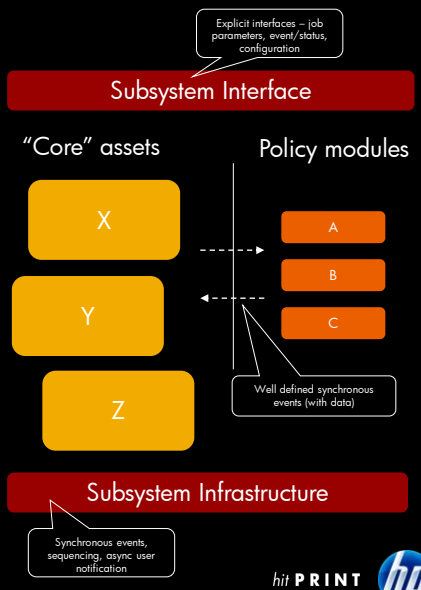
- Some high-lights…
  - Evolved from centralized "system manager" and other registration
  - Added framework support for new resource usage models
  - Enforcing of key rules
  - Converged several divergent subsystems
  - Eliminating bad patterns
  - Adopted **"policy"** pattern
- **Accomplished these whilst continuing to delivering products**

- Take aways
  - Easy to have central choke-points in otherwise decoupled system
  - Avoid "server" component knowing about clients
  - Make all component interaction explicit
  - Tackle high pain-points first
  - Phase things in when possible; but make sure it gets finished
  - Make sure infrastructure services are used appropriately…

SPLC 2009

Evolving the architecture… **policies**

- Situation
  - One component involved in all error handling for a specific subsystem
  - No explicit interfaces
  - Lots of coupling
- Approach
  - Multi-year; mostly in trunk
  - Explicit interfaces
  - Delegate pattern with a few twists
    - Decoupled, multi-receiver – aka synchronous events
  - Sequencing of actions



hit PRINT

SPLC 2009

In **summary**…

- Don't neglect tools
- Merge-capable SCM
- Establish good variation patterns
- Key agile practices
- Adapt to changes
- Evolving development model

hit PRINT

invent

Future challenges

- Modular builds / dynamic linking
- Regression testing
  - Framework/tools to make as easy as possible for component owners
- Additional complexity
  - Asymmetric multi-core, multiple embedded systems, …
- **Feature modeling**
  - Make it easy & obvious to configure/add a new project

Owen **feature modeling**…

- Want to…
  - Make it easy to add/configure a new project
  - Eliminate duplicate configuration
  - Derive other artifacts from model – e.g. project datasheets
- Restrictions
  - Maintain same dev model (single branch, all products)
  - Use CPP for variation – known model; all "paths" are visible
  - → **generate project makefile**

- Modeling **Owen**
  - Project chooses HW components
    - We typically don't put more HW than needed ☺
  - Describe high-level product features (e.g. wifi, certifications)
    - What a non-firmware manager would understand
  - FW model then
    - Depends on available HW and high-level description
    - Derives component lists, build-flags etc.
    - Describes which HW connections are required
  - A "board" object defines the logical-to-physical connections
  - Derive project makefile, header-files, datasheet, …

Modeling language

- Goals
  - Simpler than make & lazy eval
  - Be able to express modeling hierarchy
  - Automatically detect modeling dependencies
  - Smart defaults
- So far…
  - Building on top of Ruby as domain specific language
  - Have tried purely declarative
    - Difficult to learn; too much Ruby magic
  - Also tried more imperative
    - Less easy for dealing with dependencies
    - Leads to duplicate info
  - Next step
    - DSL w/encapsulated Ruby syntax
- Example – Bluetooth
  - Can be a USB dongle, built-in or not supported
  - FW need to know
    - None, dongle, built-in
  - If printer has USB host-port it's typically enabled
  - If we have a BT radio module on the board → built-in
  - DASIC needs a USB host controller

Owen model example

```
Feature.new('io.bluetooth') do
  depends_on :device, UE::Device
  depends_on :usbhost, Feature::io.usbhost
  depends_on :hw_module, HW::Bluetooth::Radio, :optional
  requires IO::BT::Profile

  attribute :support, Enumeration,
                      [:none, :dongle, :embedded]
  selection :profiles, IO::BT::Profile,
                       :conditional, :min => 1

  def support?
    support != :none
  end

  def embedded?
    support == :embedded
  end

  def process(ctx)
    default[:support] =
      if hw_module
        :embedded
      elsif usbhost.support? && device.usbfront
        :dongle
      else
        :none
      end
    values = [:none]
    values << :dongle if usbhost.support? && device.usbfront
    values << :embedded if hw_module && usbhost.support?
    validate :support, values

    derive Build::OnOff, 'FEATURE_BLUETOOTH', support?
    derive Build::OnOff, 'EMBEDDED_BLUETOOTH', embedded?

    return unless support?

    validate :profiles
  end
end
```

```
IO::BT::Profile.new('bpp') do
  requires PDL::xhtml
  requires Service::io.bt.bpp
end

IO::BT::Profile.new('hcrp') do
  requires PDL::pcl
  requires Service::io.bt.hcrp
end

IO::BT::Profile.new('spp') do
  requires PDL::pcl, :optional
  requires Service::io.bt.spp
end

Service.new('io.bt.bpp') do
  build_objmodules %w{ obex_svr }
  requires Subsystem::io.bt
end

…

Subsystem.new('io.bt') do
  build_objmodules %w{ bt_mgr service_lib bt_stack }
  configures Subsystem::io.usbhost

  def process(ctx)
    configures Subsystem::io.usbhost do |usbhost|
      usbhost.drv_bt = true
    end
  end
end
```


By evaluating the **Owen** model we get…

- A generated makefile ☺
- No project configuration (other than HW component selection)

… but we obviously have a long way to go
- Settle on the right modeling language & tools
- Model the whole system
- Phase it in

```
# Service::io.bt.bpp
OBJMODULES += obex_svr
# Service::io.bt.hcrp
OBJMODULES += io_drv_hcrp
# Service::io.bt.spp
OBJMODULES += io_drv_spp
# Service::lang.pcl
OBJMODULES += pcl
OBJMODULES += jm_pcl
# Subsystem::io.bt
OBJMODULES += bt_mgr
OBJMODULES += esi2_0
OBJMODULES += services_lib
# Subsystem::io.usbhost
USBHC_UPCOM = FEATURE_OFF
USBHC_CDR = FEATURE_OFF
USBHC_BLUETOOTH = FEATURE_ON
…
```


backup slides

Key learnings on **tools** sharing

- Share from beginning
  - Easier within single geography
  - Versioning of tools – identify which tools
    - Change w/code
    - Per project tool version
    - Suitable for SCM inclusion
    - Tools from host OS
  - Method for distributing and keeping tools up-to-date across geographies
- Examples
  - Compiler version per project
  - Specific version of tool not available on OS
  - Tightly coupled to code
- What we did (circa '04)…
  - /owen/tools NFS mount
    - rsync across geographies
    - Separate SCM repo for tools; deploy using rsync on commit-hook.
  - /owen/tools/bin
  - /owen/tools/<vnd>/<version>/…
  - Build picks appropriate tools based on project config
  - Common Linux setup
- Others
  - Same defect tracking & requirement system