Randy Marques Consultancy
Embedded Software Development

GBS
Generic Build Support

A Pragmatic Approach to the Software Build Process

6.00

randy.marques.com@xs4all.nl

Introduction - Who am I

- Randy Marques - CASE Consultant
  - Retired
  - CEO / Owner Randy Marques Consultancy
  - Nederlands Normalisatie Instituut (NEN)
    - Nederlandse Programmeertalen Commissie (NC 381 22)
      - WG14 (International ANSI-C Committee)
  - Teach at various Universities and Colleges

- "Consultancy by Walking Around"
  - Software Engineering since 1971
  - Coding Standards since 1978
  - Build Automation since 1980
  - C Programming since 1983
  - Static Analysis since 1993
  - Les Hatton's Safer C™ trainer since 2001

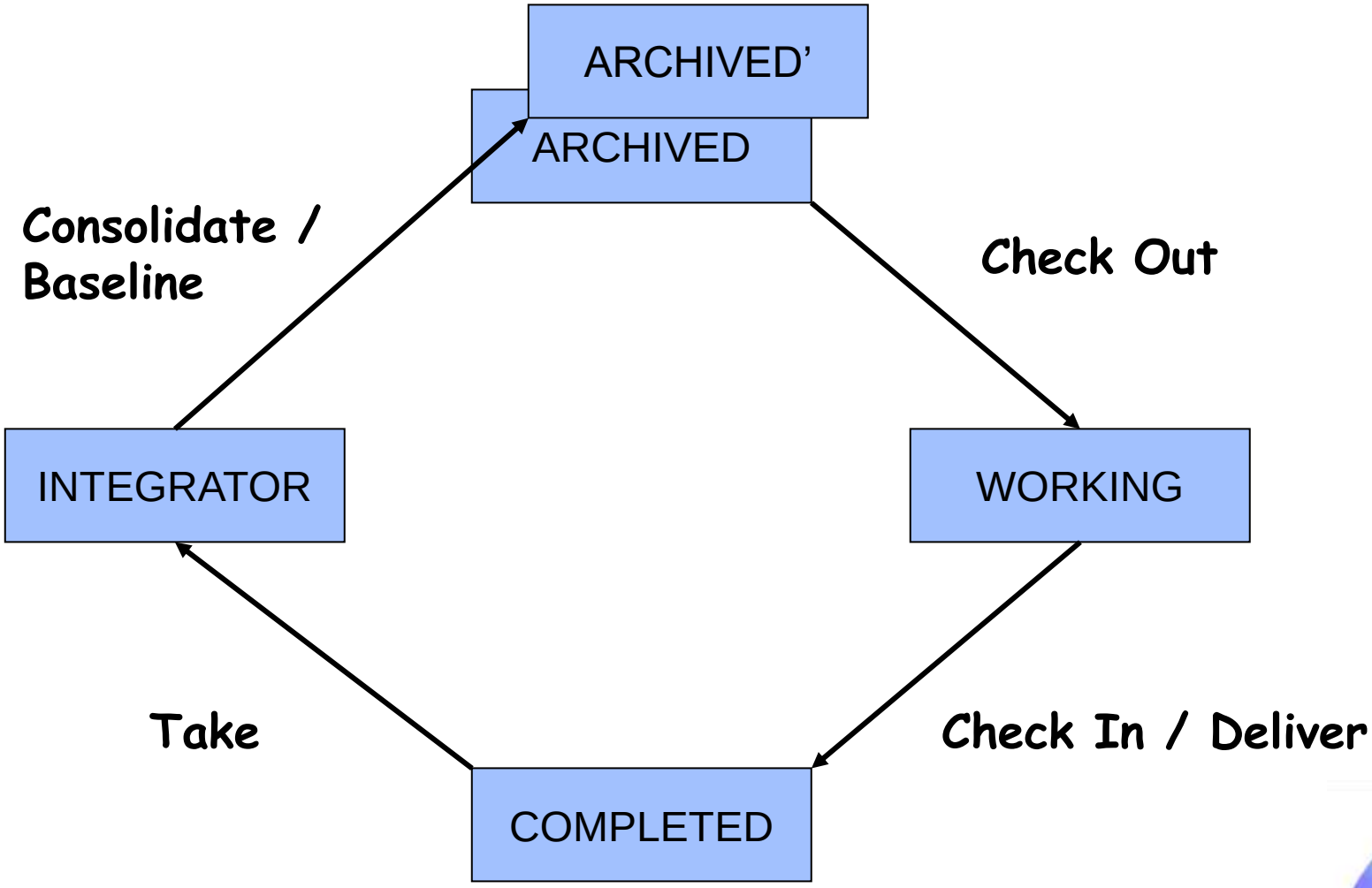
Scope

- Low, Low Level ‘Configuration Management’
- Build Automation
  - Compiling, Linking
  - F5
- Issues
  - Directory Structure
  - Scripting (‘make’)
  - Multi Platform Variants
  - Functionality Variants
  - 3rd party software, Subcontracting, ‘re-use’
  - Releasing

Overview

- Task Lifecycle
- Problem Areas
- The Solution
- Does it Work
- Final Remarks & Questions

Task Lifecycle:
Single Level Promotion Model



Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Volume

- Software Generation:
  - Lines of code
    - 20 files of 100 lines -> 5000 files of 400 lines
    - 2000 lines -> 2.000.000 lines
      - Generation takes better part of the night
    - 2 ‘specials’ (10%) -> 500 ‘specials’
      - Quick Fix???
  - People
    - 4 programmers -> 80 programmers
    - 1 Programmer
      - 1 Check-in every week
      - 1 build-error every 15 check-ins
        - » Every 75 days 1 build-error
    - 75 Programmers
      - Every day the build fails!

Volume

- Success breeds complexity
  - Brian A. White
    (Software Configuration Management Strategies)

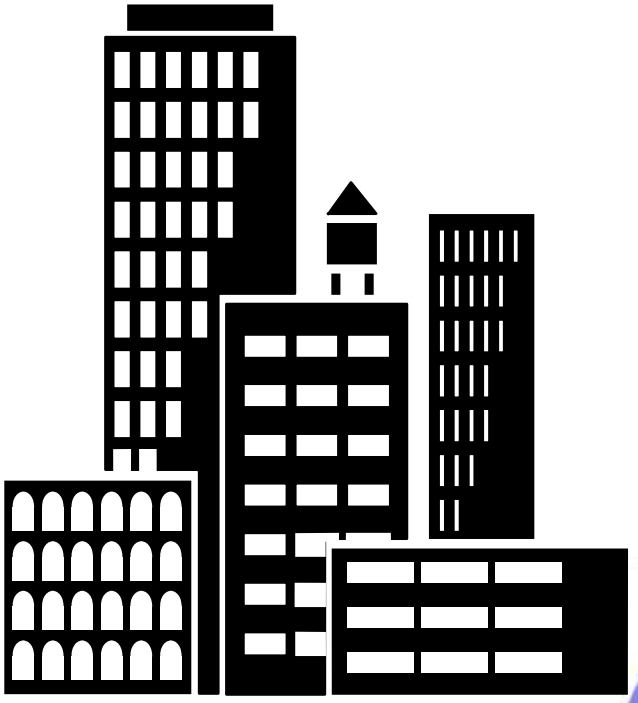
- The size of code in consumer electronic products currently doubles every 18 months
  - Line-Scan TVs have ~ 1.500.000 lines of code

- Murphy's Law:
  - If something CAN go wrong,
    if you do it often enough,
    it WILL go wrong

From Barn to Skyscraper

- We started off building yard barns

- And we are convinced that to build skyscrapers we only need to extrapolate...



Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Problem Area: Scripting

- Start simple
- Complex and elaborate scripting
  - Dedicated
    - per project (subsystem / component / file)
    - per platform, per user, per exception
    - per release
  - Used to solve Managerial and Architectural issues (Silver Bullet)
  - Support for the weirdest exceptions
    (even the one we consider 'bad practice')
  - Inconsistent
  - Platform dependent
  - Impossible to change
  - Long learning curve (use and maintenance)
  - Much, very much maintenance
  - Hard-coded, absolute file-paths, search-paths
  - Dependent on external local settings (.profile / Registry)
  - Abuse of 'make'

Problem Area: ‘make’-Scripting

- The essence of ‘make’:
  - re-generate as little as possible
  - Dependencies
  - ‘A Program for Directing Recompilation’ (GNU-make)
- Pollution of ‘make’
  - Used to define build-process instead of supporting build-process
    - Header-file directory specification
    - Compile options
    - Generation of various functionality/platform variants
  - Unsafe make:
    - Always 'clean' and 'build all'

If I always regenerate my whole system
(because my 'make' is not safe):
Why do I use 'make'?

Problems With ‘make’

- Maintaining Dependencies
- Start-off small & simple.
  Grow and become too complex
  - No-one dares to make modifications
  - Not-maintainable make-files (> 40.000 lines!)
- Multiple checkout on make-files
- Completely platform/environment dependent
- Build for more than 1 platform?
  - On different machines
- Suffix rules: all files must be in same directory
- Search-paths (differ on various machines)
- more...

Problems With 'make'...

- Recursive make
  - Recursive make Considered Harmful (1997)
    Peter Miller
    http://www.canb.auug.org.au/~millerp/rmch/recu-make-cons-harm.html
- Weird restrictions: e.g.:
  - only 1 library/executable per component
  - make of single file not possible
  - etc.
- Extremely complex

No Design, No Standards, No Reviews,
No-one feels responsible

‘make’ Basics

- make-File:
  - Collection of make-rules
  - Macros & Control-statements

- make-Rule:

```
target : dependencies
<TAB>  command [; …]
```

```
edit.exe : main.o kbd.o command.o
<TAB>      lnk -o edit.exe main.o \
               kbd.o command.o
main.o : main.c defs.h
<TAB>    cc -c main.c
kbd.o : kbd.c defs.h command.h
<TAB>   cc -c kbd.c
command.o : command.c defs.h \
            command.h
<TAB>       cc -c command.c
```

What About IDEs?

- IDEs:
  - Visual Studio, Tornado, Greenhills
  - OpenSource: ant, scons, quickbuild, etc.
- Problems:
  - Obscure internals
  - Dedicated:
    - No interaction between different brands
    - Build for more than one Platform: problems
  - (Elaborate) scripting needed
  - Based on ‘handy-tool’ concept. Not SWE concept.
    - Flexible, so every project (release) has different implementation…
  - Sales argument:
    "We will support whatever way of working you want“

What About IDEs?

- Make life simple on programmer's level
  ("Every nitwit can")
  not on Project or Company level.
- No support for structured way of working (scoping)
- Limitations on number of executables, libraries, etc
- Dependent on search-paths
- No automatic testing integration (module-test)
- Dependent on various specific versions of different tools (OpenSource)
- Anyone can do it 'in his own way'

Overview

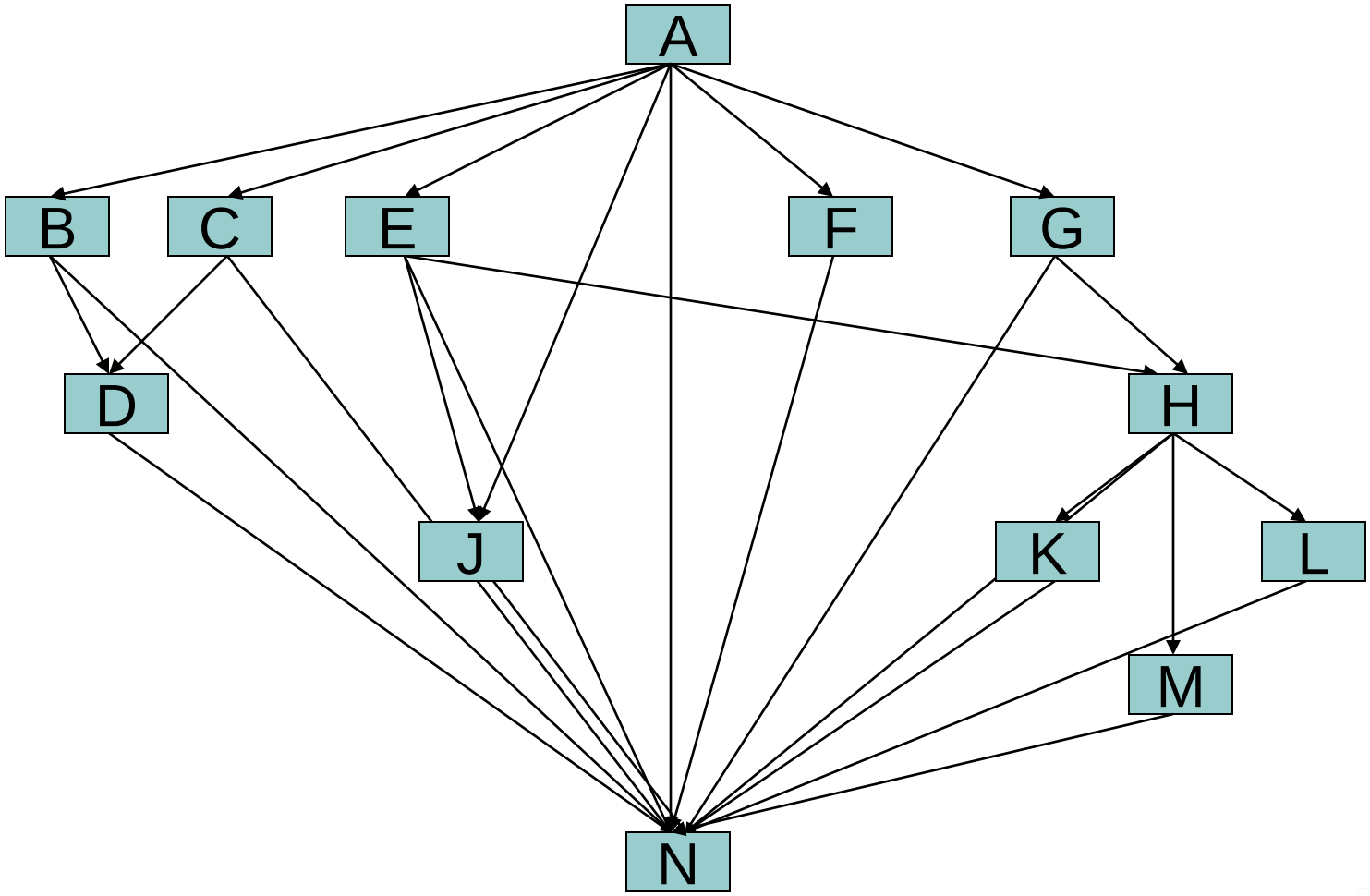
- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Problem Area: Directory Structure

- Trying to map a software-structure on a directory structure
  - Software Structure:
    - Tree-like Network Structure
  - Directory Structure:
    - Tree Structure

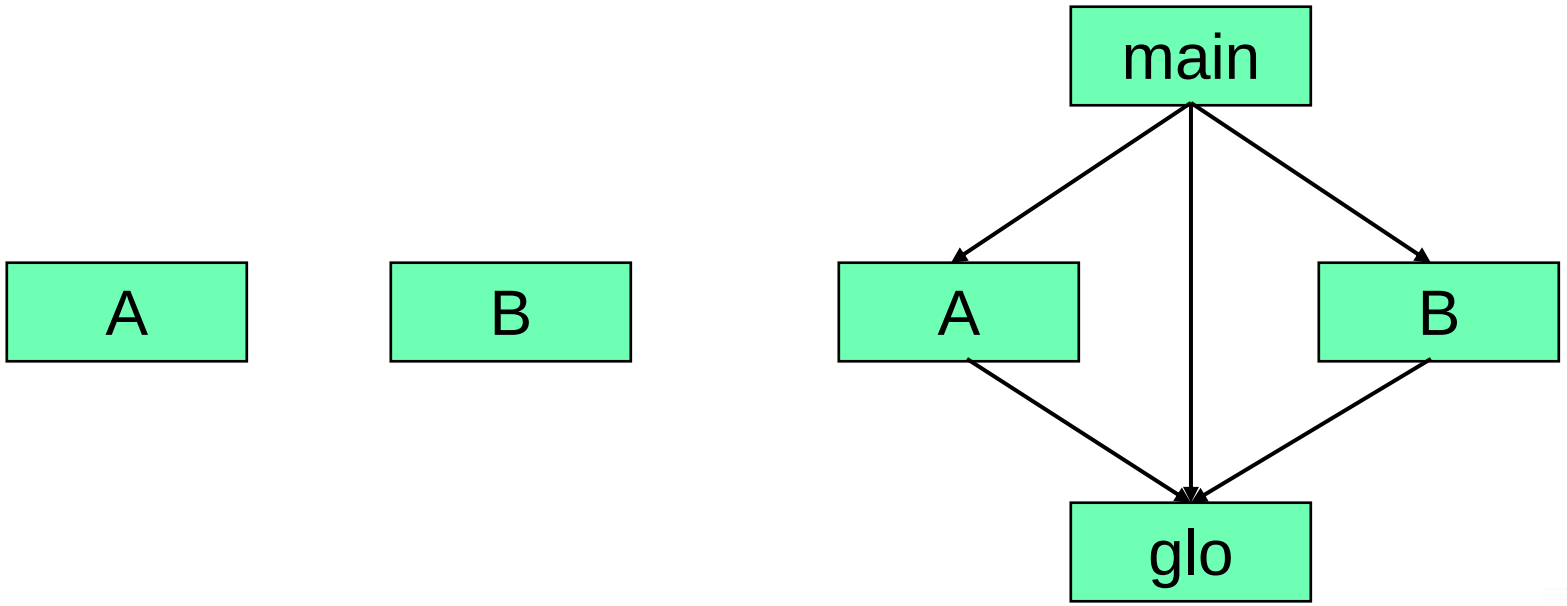
- Relocating a directory inside the tree structure causes major problems in generation ('make') files

Software Structure



Component Structure

- A multi-component system always consists of at least 4 components:



Directory Structure - Flexibility

- What effort does it cost to add a new component
  - or compiler
  - or variant
  - or platform
- What are the consequences for an SCM-tool
- What do I do with ‘re-use’ from another project
- Environment variables, System Settings
  - `.profile`?
  - Registry?

Do I spend 4 man-months to define my dedicated directory-structure and 'make'-method every time I start a new project?

Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Problem Area: Integrity

- How safe/complete is my generation process
  - Was really everything compiled before the link?
  - Did all files compile properly before Freeze/Archiving/Release?

- Given an executable, can I recreate exactly the environment leading to its creation?
  - Sources
  - Parameters (compile-time options)

Given the amount of money spent on SCM-tools and procedures, how reliable is my ability to regenerate the software?

Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Problem Area: Variants - 1

- Versions
  - Taken care of by SCMS
- Variants / Diversity
  - Platform
  - Functionality
    - Hardware
    - Software
- Compile-time Options & Conditional Compiles
  - Many Options
  - Many Possible Values

```
#ifdef __SUN__
  …
  …
  …
 #endif
```

Problem Area: Variants - 2

- Problems:
  - Too many permutations
    - Always starts with just a few...
    - Which combinations are valid?
    - Not readable, not maintainable
  - Which variant is now in my object-library?
  - Which variant did I build my executable with?
  - Compile archived version with different option:
    - compilation fails
  - Not Testable

Given an executable that was built on a specific date and time:
What were the settings that led to this executable?
(I mean: ALL the settings...)

Platform Variants

- Compile-Time diversity (Conditional Compiles)
- Link-time diversity (Multiple Sources)
  - file.c
    - file_pc.c => file_pc.o `time()`
    - file_vms.c => file_vms.obj `time()`
    - file_sun.c => file_sun.o `time()`
  - BLD
    - PC <= Build-output Directory
    - VMS
    - SUN
- Combinations

Platform dependent functionality should be isolated into one or two components

Functionality Variants

- Link-Time diversity (Multiple Sources)
- Run-Time diversity
- Combinations

- As an example we will use the generation of two executables that share a lot of code:
  - A Video-Player and
  - A Video-Recorder

Compile Time Diversity

- Traditional Approach:

```
MAKE-FILE:
   -D RECORDER

FILE.C:
   #ifdef RECORDER
      …
      …
    #else
      …
      …
    #endif
```

Run Time Diversity

- Compile-Time:

```
#ifdef RECORDER
   ...
   ...
   do_recorder();
#else
   ...
   ...
   ...
#endif
```

- Run-Time:

```
if (recorder())
{
   ...
   do_recorder();
} else
{
   ...
   ...
}
```

Run Time Diversity

- Setting the condition for ‘recorder’:
  - Link-Time diversity
    - File1: return TRUE
    - File2: return FALSE
    - Link with either file
    - 'Stubs' to save memory
  - Read a file (.ini)
  - Read Hardware jumpers
  - Read command-line arguments
  - Ask the user

Diversity is not a Build issue:
It is a SW Architectural issue

Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Delivering

- What do I deliver
- What is my product
  - Executable
  - Libraries + headerfiles
  - Directory structure
  - Install
- How do I “assemble” my deliverable?
  - Copy directly from bld-directory?
  - Document?
  - Script?

Overview

- Task Lifecycle
- Problem Areas
  - Volume
  - Scripting
  - Directory Structure
  - Integrity
  - Variants
  - Delivering
  - Tooling
- The Solution
- Does it Work
- Final Remarks & Questions

Problem Area: Additional Tooling

- How to implement additional tooling
  - Doxygen
  - SCAs like PCLint, QAC, C++Test
- Automatic Testing
- Multiplatform?
- Example:
  - QAC
    - How do I specify which files to check
    - How do I obtain header-file include directories (`-I`)?
    - How do I obtain compile time options (`-D`)?
    - ‘make’ ?

Overview

- Task Lifecycle
- Problem Areas
- The Solution
- Does it Work
- Final Remarks & Questions

Solution

Generic Build Support

- A Concept
- A Set of Agreements
- A Rigid Directory Structure
- No built-in Knowledge of Application
- No user-written scripts
- Generated 'make' Files
- Multiplatform generation
- Implementation Support:
  - Scripts
  - Currencies
    - Current System, Current SubSystem, Current Component, Current Build

Solution: Basic Rules

- Simple approach. No Black Box. KISS
- Transparent: No clever tricks
- Rigid where no flexibility is required
  Full flexibility where flexibility is needed
- Do not cater for things we agree to be ‘bad practice’
- No project-dependent scripting
- Automate anything that can go wrong when done manually
  (but do not try to achieve 100% automation when in conflict with Rule 1)
- Must be generic (no knowledge of application)
- Must be SCMS and target-platform independent
- Must have a relocatable directory-structure

Prepare Yourself

- Take a deep breath
- Stretch yourself
- Empty your mind
  - Do NOT think of your current implementation
- Relax

Overview

- Task Lifecycle
- Problem Areas
- The Solution
  - Directory Structure and scripting
  - Integrity
  - Finalizing
- Does it Work
- Final Remarks & Questions

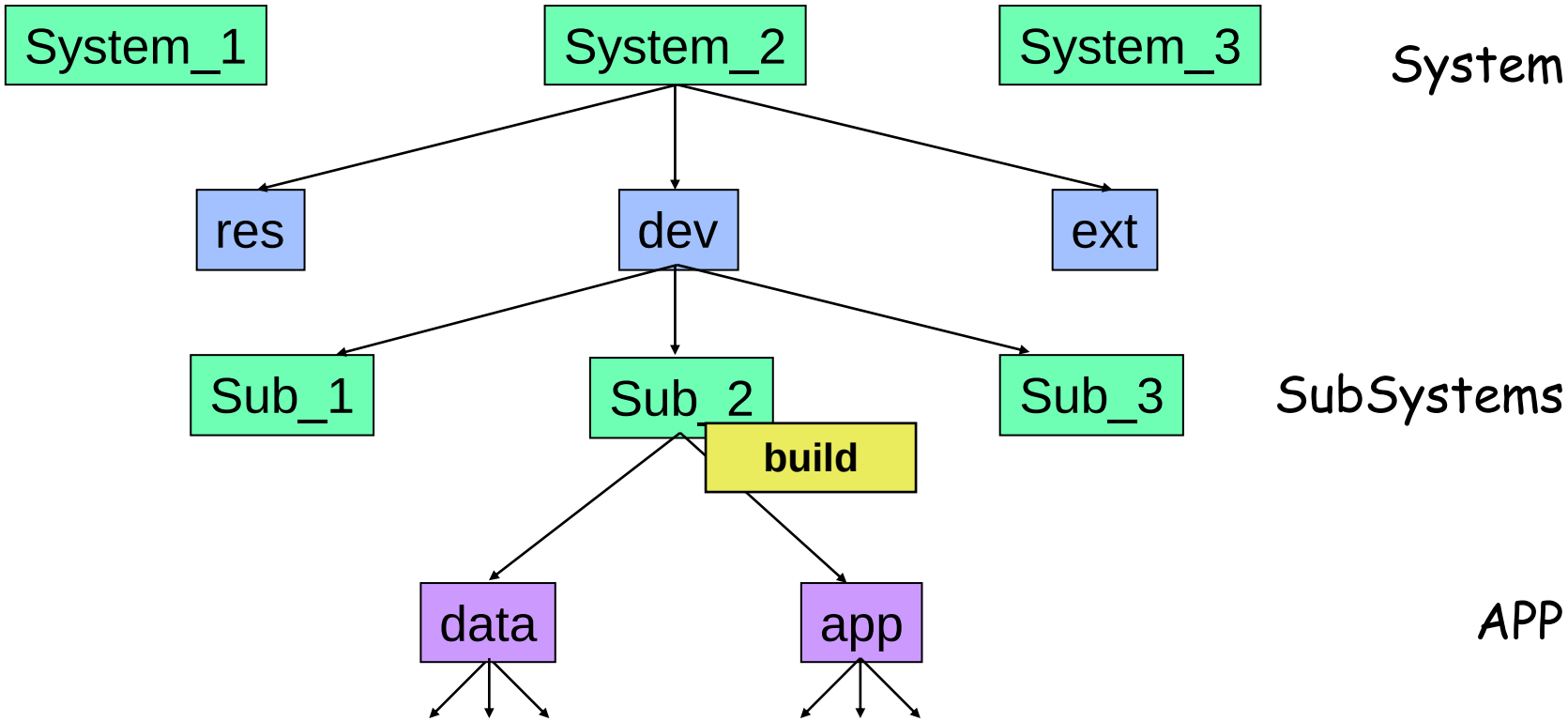
‘Flat’ Directory Structure

- System:
  - EXT (externals) Directory
    - 3rd party SW Directories
  - DEV (Development) Directory
    - SubSystem Directories
  - RES (Results) Directory
    - SubSystems Transfer Directories

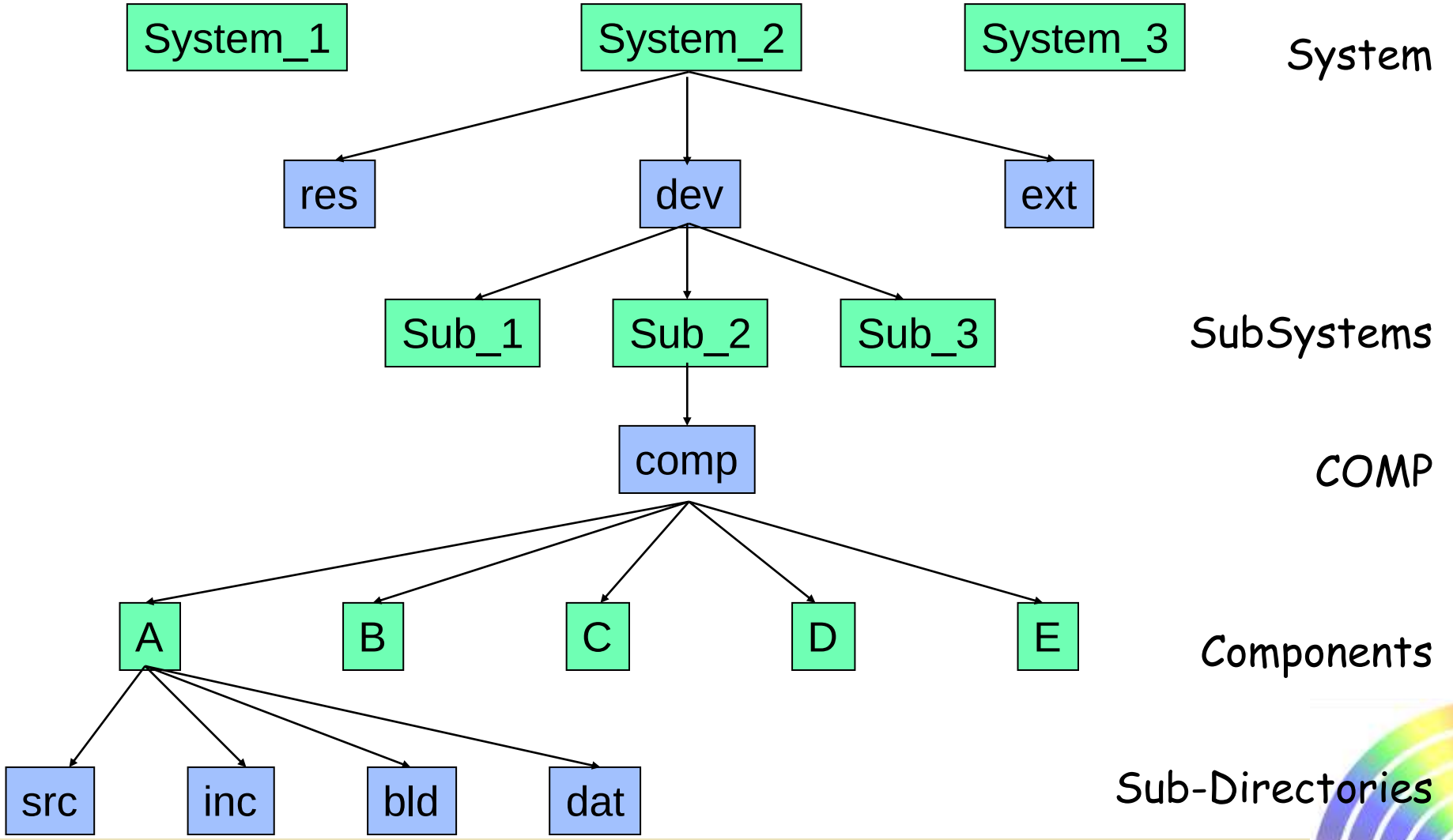
‘Flat’ Directory Structure

- SubSystem Directory (non-GBS Compliant)
  - APP-Directory
    - Grow Your Own
  - DATA-Directory
    - Grow Your Own
  - EXPORT-Directory (Optional)
  - Main scripts
- SubSystem Directory (GBS Compliant)
  - COMP-Directory
    - Component Directories
      - Sub-Directories (SRC, INC, BLD, etc.)
  - EXPORT-Directory (Optional)

Non GBS Compliant



GBS Compliant



Sub Directories

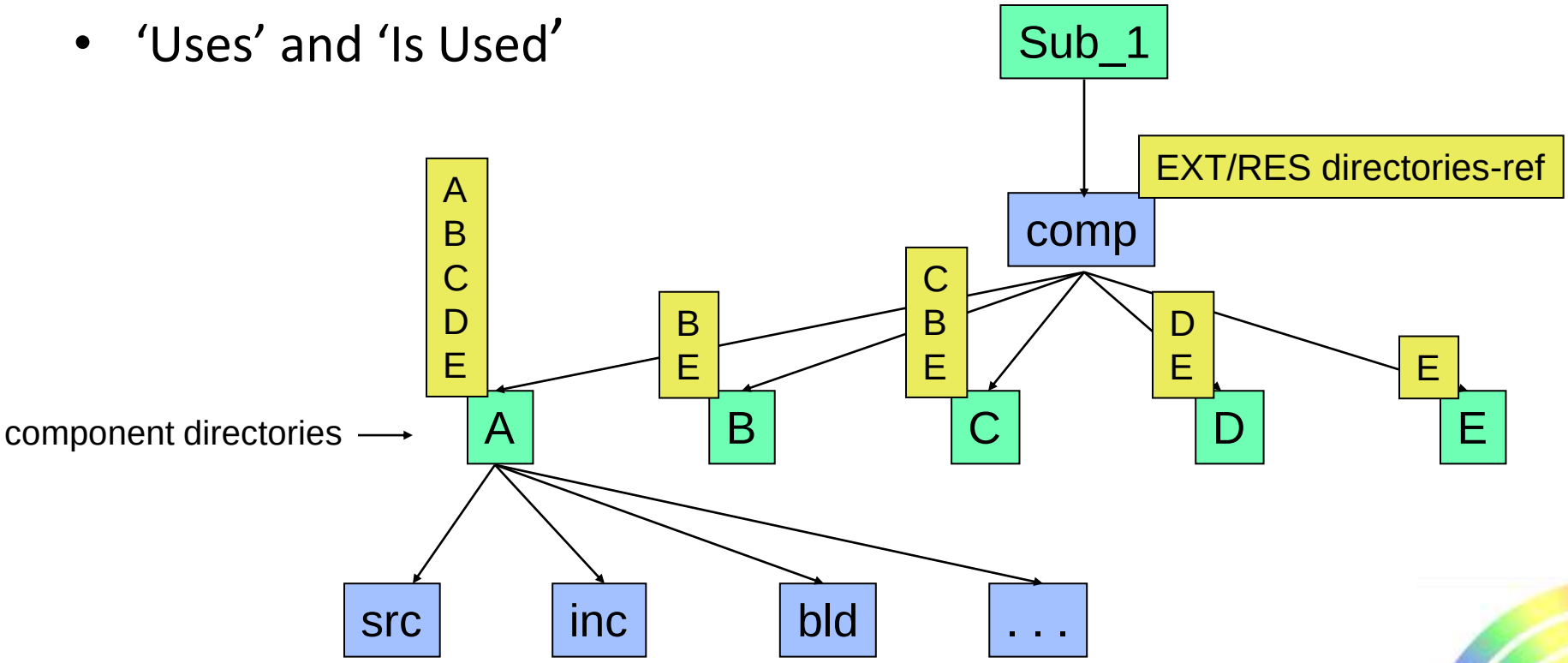
- SRC
  - Sources
- INC
  - Global (exported) Header-files
- LOC
  - Local Header-files
- BLD
  - Contains <build>-Directories
    - Results of building (compilations)
- AUD
  - Results of additional tooling (qac, pclint etc)
- *More later...*

Currencies

- GBS works with ‘currencies’:
  - Current System
  - Current SubSystem
  - Current Component
  - Current Build
- Navigate thru the directory tree by means of ‘switch’ commands:
  - `swr`: switch System
  - `sws`: switch SubSystem
  - etc.
- The current Build specifies a specific compiler environment
  - Compiler / Archiver / Linker to use
  - Settings

Scoping

- Scope-files contain component-names, no directory specs.
- SCOPE.GBS in Component Directory
  - Specifies the 'VIEW'
- 'Uses' and 'Is Used'

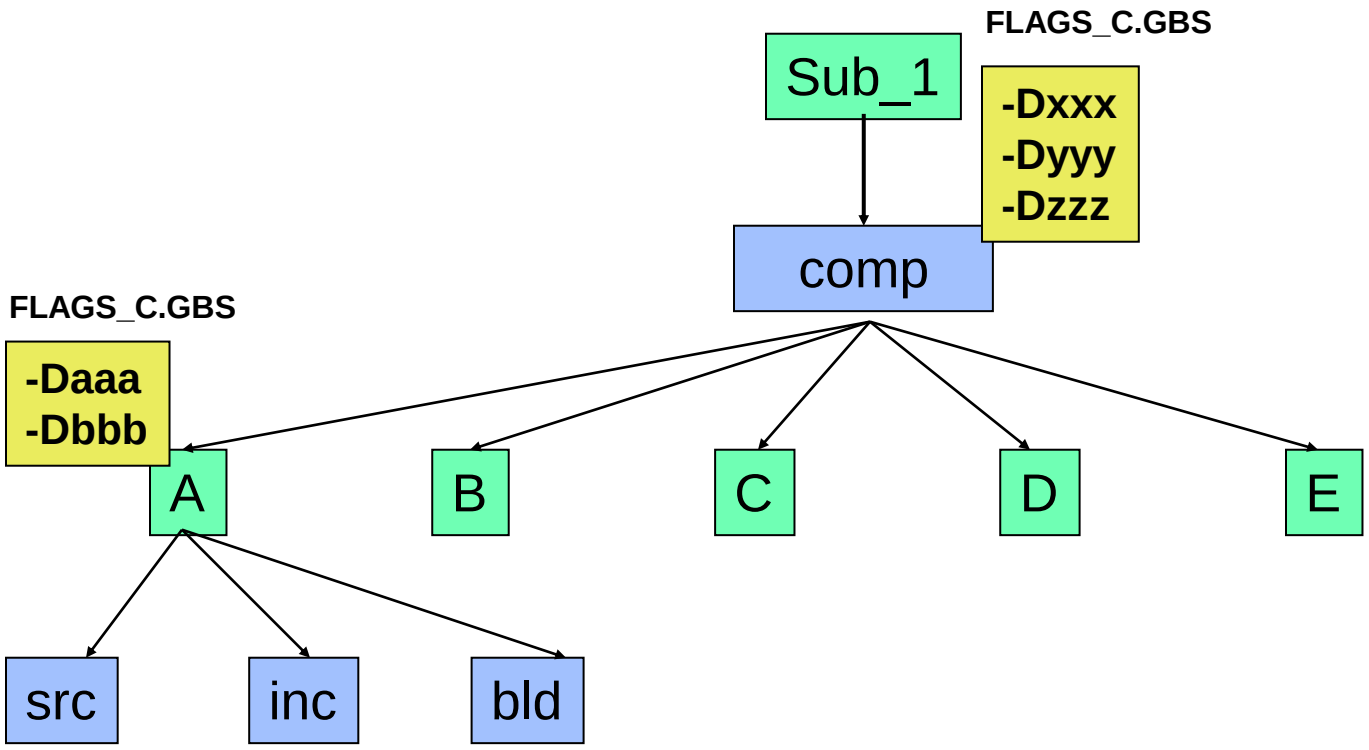


Compile-time Options - 1

- Options are placed separately in option-files
  - Options for all C-files in project:
    - FLAGS_C.GBS In COMP directory
  - Options for all C-files in component:
    - In FLAGS_C.GBS in Component directory

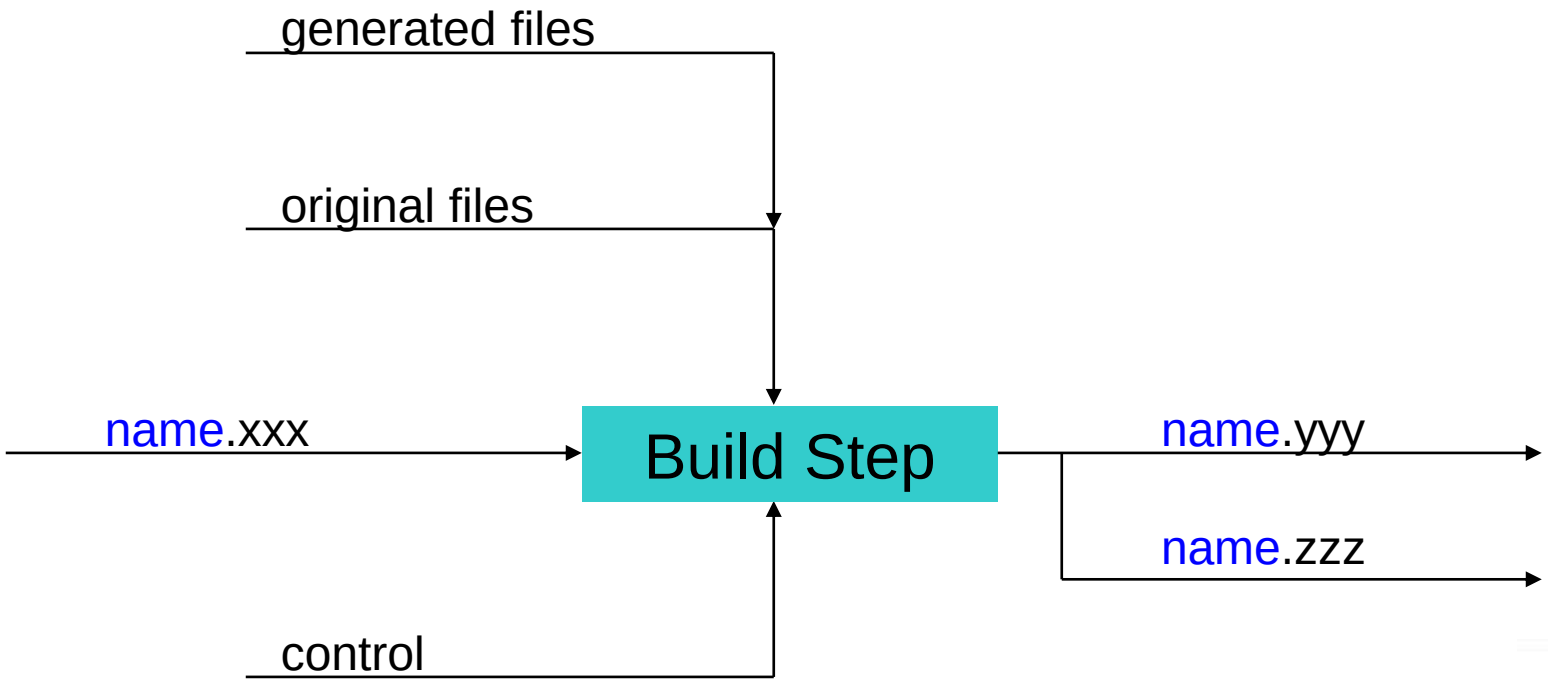
- For compilation, options are placed in the order specified above.
  - Last option wins...

Compile-Time Options - 2



Build Automation Basics

- Anatomy of a Build Step



Generating a Compilation

- Given a source file of the current Component of the current SubSystem of the current System with a current Build, we have:
  - Source File name (*file*.c)
  - Compiler to be used
  - Extension of object-file name (.o)
  - Object-file name (*file*.o)
  - Header-File Directory information
  - Compile Options Information
  - Input & Output Directory
- So we have a generic script that generates a dedicated compile command.

Libraries And Executables

- What if I do not have a source-file?
  - Libraries
  - Executables

Generating a Link

- Traditionally done in ‘make’ file

- Enter: Link-file
  - Works the same way as 'compile file'
  - *name*.glk => *name*.exe
  - Contains:
    component:*objectfile-name*
  - Also:
    .include <*inc-filename***>**

- So we have a generic script that generates a dedicated link command.

- *name*.glk => *name*.exe
- Contents:

```
A:main.o
A:a.o
A:a1.o
B:b.o
B:b1.o
C:c.lib
```

Generating a Library

- Traditionally done in ‘make’ file

- Enter: Library-file
  - Works the same way as 'compile file'
  - *name*.glb => *name*.lib
  - Contains:
    component:*objectfile-name*
  - Also:
    .include <*inc-filename***>**

- So we have a generic script that generates a dedicated archive command.

- *name*.glb => *name*.lib
- Contents:

```
A:a.o
A:a1.o
B:b.o
B:b1.o
C:c.lib
```

Automatic Testing too…

- Module-tests as part of the build-system
- Enter: Test-file:
  - Works the same way as ‘compile file’
  - *name*.glt => *name*.log
  - Contains:

    `component:executable-name`
  - Returns: Normal or Fail

- So we have a generic script that generates a dedicated test-command

- *name*.glt => *name*.log
- Contents:

```
A:MyTest.exe
```

Generating a SubSystem

- In the SRC directories of all Components:
  - Compile all *.c files into objects
- In the SRC directories of all Components:
  - Archive all *.glb files into libraries
- In the SRC directories of all Components:
  - Link all *.glk files into executables
- All results go to the current BLD/<Build> directories

Flavors of 'build'

- gbsbuild file.c
- gbsbuild <list of individual files from same or other component>
- gbsbuild <all files in one or more components>
- gbsbuild <all files in one or more subsystems>
- gbsbuild *:*.*
- etc...

Generating a make-file

- For each Component:
  - Take each file in the SRC-directory as a 'make'-target
  - Parse the file using the same information as for generation of Compile, Archive or Linking to determine dependencies
  - Generate the make-file

Flavors of ‘make’

- gbsmake image.exe
- gbsmake <list of specific files>
- gbsmake <all files in list of specific components>
- gbsmake <SubSystem>
- etc...

Differences between 'build' and 'make'

- 'build'
  - you specify the source (e.g. file.c)
  - **only** the specified file(s) will be built
  - **all** the specified files will be built
- 'make'
  - you specify the resulting file (e.g. file.o)
  - other files (even in other components) **may** be built
  - specified files **may** or **may not** be (re-) built

Additional Tooling

- Given the method for generating a dedicated build step command we can easily implement any kind of auditing (SCA) tool
  - QAC, QAC++, PCLint , C++Test
    - Implemented
  - etc.
  - The command for this is gbsaudit
  - Results go to the aud/<audit>/<build> directory

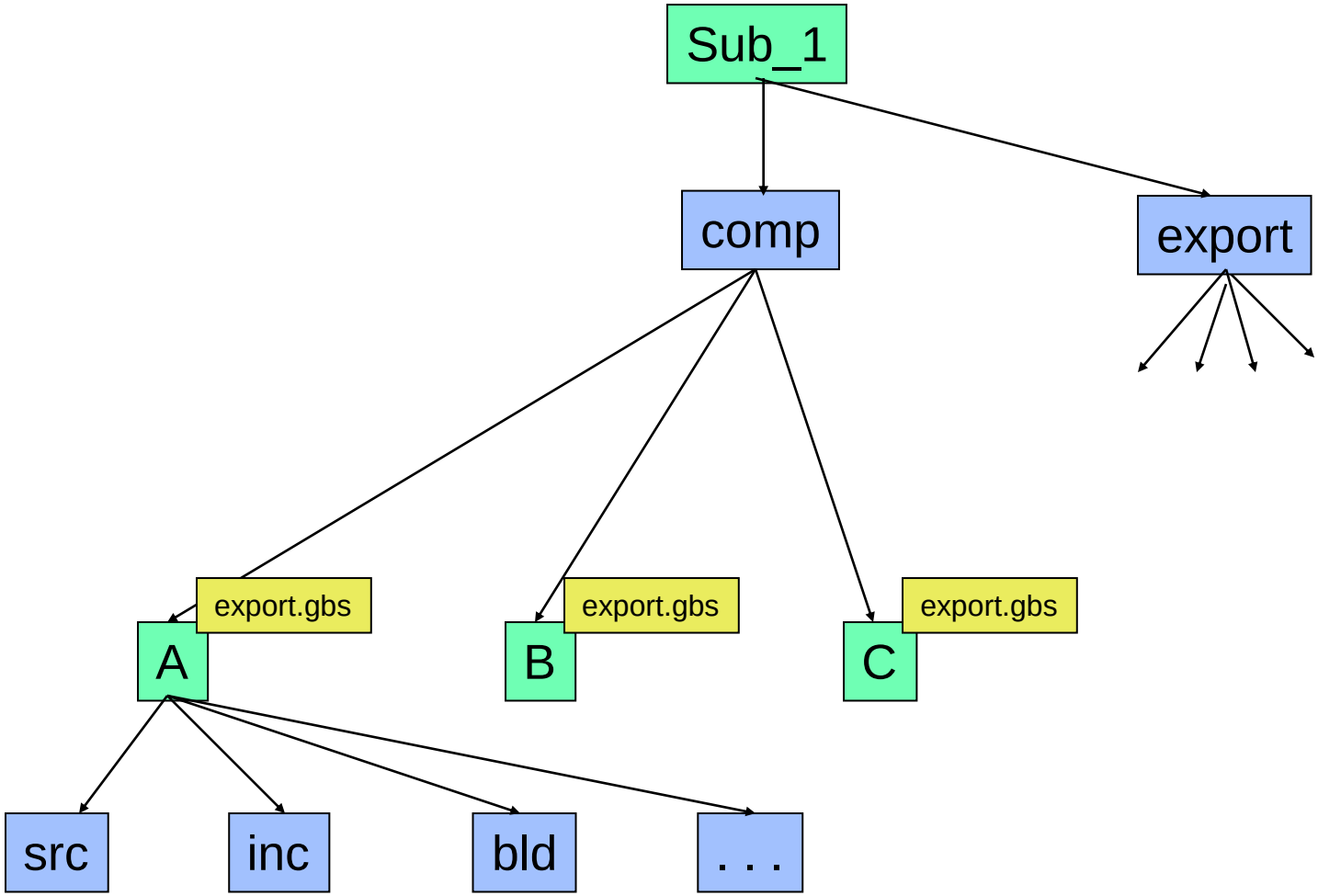
Additional Tooling (System wide)

- Also tools like DoxyGen (Implemented) that work more on System level.
  - The command for this is ‘gbssystool’
  - Results go to the silo directory

Creating Deliveries

- Export directory
  - Results for the end-customer
  - Any set of (sub-)directories
- export.gbs files
  - One per component
  - Specifies which files to export
  - Build dependent
- gbsexport

Creating Deliveries



Overview

- Task Lifecycle
- Problem Areas
- The Solution
  - Directory Structure and scripting
  - Integrity
  - Finalizing
- Final Remarks & Questions

Verifying Builds

- When the generation of a file fails the possibly generated file in the BLD will be deleted.
  - In case of compilation or archiving, subsequent steps (e.g. link) will also fail
- Before CONSOLIDATION we can now check whether a file generated properly:

  If there is no corresponding file in the BLD directory, the CONSOLIDATION may NOT be executed.

SRC:<name>.<type>

must match

BLD:<name>.*

Overview

- Task Lifecycle
- Problem Areas
- The Solution
  - Directory Structure and scripting
  - Integrity
  - Finalising
- Final Remarks & Questions

Subdirectories - 1

- SRC
  - Trigger in generation process. Results go to BLD/<build> directory
  - Scanned for dependencies.
  - Archived in SCM
- INC
  - Exported to generation of other components
  - Scanned for dependencies.
  - Archived in SCM
- LOC
  - Scanned for dependencies.
  - Archived in SCM
- BLD
  - <build> Contains results of SRC generations
  - Exported to build of other components
  - Scanned for dependencies
  - Full delete of contents allowed

Subdirectories - 2

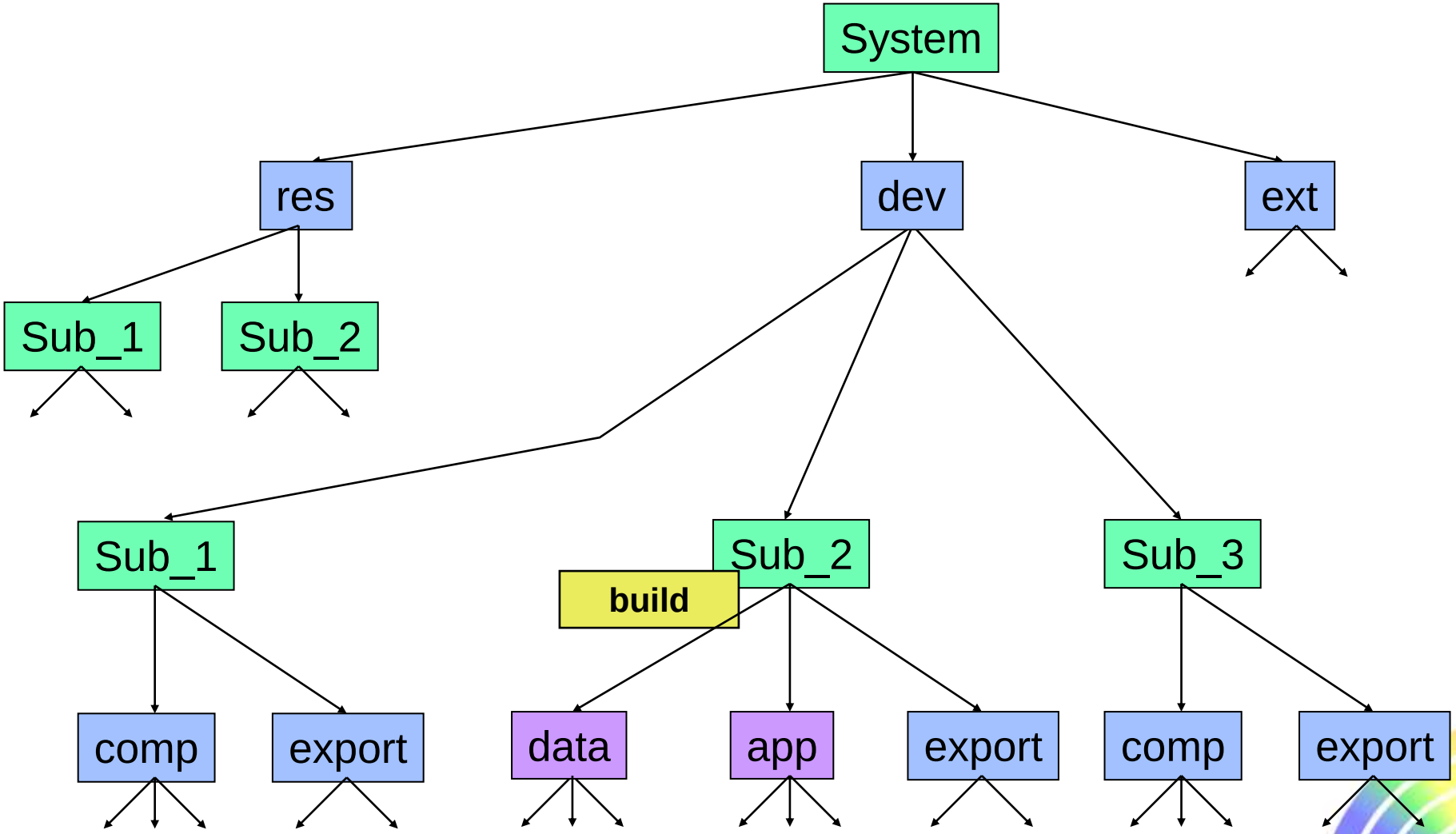
- AUD
  - <aud>/<build> Contains results of 'audits' (QAC / PCLint, etc)
  - Full delete of contents allowed
- DAT
  - Contains any other data that partakes in the assembling (not building) of the resulting delivery. e.g.:
    - bitmaps
    - scripts to run/test the deliveries
  - Archived in SCM
- SAV
  - Contains anything that has to be kept but does not partake in the generation-process, at this time
    - old stuff
  - Archived in SCM

The Bigger Picture - 1

- Handling of SubSystems on a System Level
  - export directory
    - deliverables
    - export.gbs file in every component
  - res directory
    - subsystem directory
  - gbsexport
- Generation of non-compliant SubSystems
  - export directory
  - 'build' script

- Direct reference from one SubSystem to another is absolutely not allowed

The Bigger Picture - 2



The Bigger Picture - 3

- To build a whole System:
  - Build first SubSystem
    - GBS-compliant:
      - 'build' or 'make' the SubSystem
      - 'gbsexport' (copies files to 'res'/'export' directory)
    - Non GBS-compliant:
      - Execute 'build' script in SubSystem directory
        At the end, deliverables must be in 'export' directory
      - 'gbsexport' (copies files to 'res'/'export' directory)
  - Build next SubSystem
    - etc

Overview

- Task Lifecycle
- Problem Areas
- The Solution
- Does it Work
- Final Remarks & Questions

Does it Work

- GBS is based on experiences with 'CCS', the Code Control System developed at Philips Medical Systems in 1985
  - Designed and developed by Randy Marques
  - Used for over 15 years by a variety of small and large projects
    - still in use for maintenance projects
  - Written for VAX/VMS in DCL
- GBS Started development in 2001
  - New technology – same concept
  - 2013: Improved User-Interface
  - 2018: GUI
- Now available for Windows 10, WSL and Linux
  - Written mainly in Perl to ensure **exact** same functionality on all platforms
  - Only dependent on Perl. No special privileges required.

ASA-Lab Pilot - 1

- Pilot project at Philips ASA-Lab
  - Multi-Site (Eindhoven, Dublin, Suresnes and Bangalore)
  - Approx. 30 developers (UNIX + NT)
  - Processors: ST20, MIPS5.1, PC
  - Compilers: OpenTv (1), C (3), C++ (3), Assembler(2)
  - SCMS: ClearCase UCM
  - Application: Multi Language variants: 5 x 2 Executables
  - Size
    - 13 SubSystems
    - 241 Components
    - 1182 Source-files
    - 611 Local header files
    - 545 Global header files
    - 160 Libraries / 41 Executables
    - 8 Externals (3rd party components)

ASA-Lab Pilot - 2

- Originally:
  - Both platforms: existing code in separate archives
  - Considerable amount of shared code
    - Good idea: Universal Archive
      (let same code go through 2(3) compilers)
  - Considered major effort:
    - 6 man-months in 3 months
    - Non-workable situation during 1 month for 10 people
  - 2x5x4=40 executables from 20 full generations
- With introduction of GBS:
  - First platform: silently
  - Second platform:
    - 1 man-month in 3 weeks
    - Same way of working for all platforms:
      - No impact for others during introduction
  - 2x5=10 executables from 2 full generations
  - QAC fully integrated

ASA-Lab Pilot - 3

- Originally:
  - Developers:
    - Had no idea of how the build (make) process worked.
    - Did not know how to introduce new (source-)files
    - Did not know where to find headerfiles and/or libraries
    - Did not know how to generate a new library or executable
    - Did not know how to generate another SubSystem
    - Could not run QAC
  - The 'make' process was unsafe
- With introduction of GBS:
  - Developers:
    - Know exactly where to place and find files
    - Understand the 'make' process
    - Agree: More output per worked hour
  - Safe 'make' process

Facet - 1

- Project at Atos Origin for Electron Microscope
  - 4.5 developers (MSWindows)
  - Processors: PC
  - Compilers: Visual C, MFC, COM (idl, mc and rc-files)
  - SCMS: Visual Source Safe
  - Size
    - 1 SubSystem
    - 11 Components and growing
    - 152 Source-files
    - 300 KLOC
    - 2 Externals (3rd party components)

Facet - 2

- Benefits
  - Starting-up GBS took 1 day
    (after 1 week of investigating the Microsoft build environment…)
  - Minimal (no) overhead when introducing new component or files
  - Hardly any build problems
  - 6 hours/week build-overhead reduced to less than 2 hours/week
  - Consistent and reliable build-environment
  - Very short learning curve
  - Command-line interface (fast!)
- Drawbacks
  - Command-line interface (learning curve)

Where is (was) GBS used

- Philips AppTech, Eindhoven
  - Various projects
- Philips Lighting, Eindhoven
- Philips Research, Eindhoven
- IWEDIA, Rennes
- TASK24/Nspyre, Eindhoven (SAS)
  - Various projects
- SiemensVDO
  - Eindhoven
  - Budapest
  - Rambouillet
  - Bangalore
- Rialtosoft (Eindhoven)

Current State

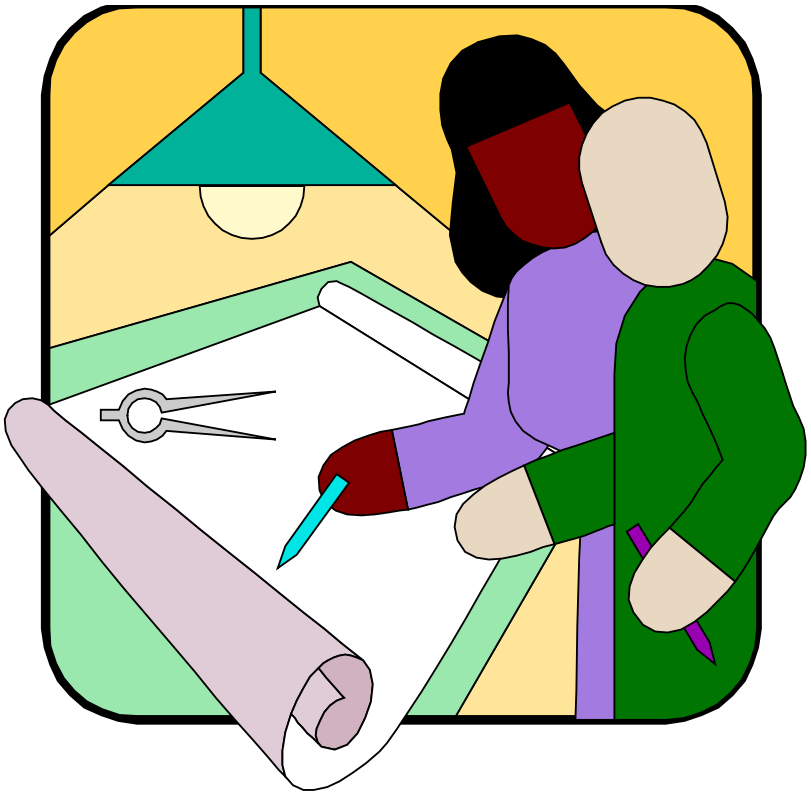
- Measurable benefits:
  - Move from ClearCase to Synergy
    - GBS part: 30 minutes
    - Non-GBS part: 2 weeks
  - From PDSL Eindhoven to IWEDIA Rennes
    Same software as above
    - GBS part: 1 hour
    - Non-GBS part: 1 week
  - Philips AppTech
    - Add a new compiler: 15 minutes
    - Implement LSF: 10 minutes
- General benefits:
  - Quick project startup
  - Interchangeable developers and components
  - "Build problems are not an issue"

Overview

- Task Lifecycle
- Problem Areas
- The Solution
- Does it Work
- Final Remarks & Questions

Good Design Poor Masonry…

- Design gets lots of attention. (Architect)



- (Too) Little concern / appreciation for the building process.
- Code Architect (Construction Supervisor) (Dutch: bouwmeester)

Features

- Fully portable and relocatable directory structure
- Multiple platform support (Win10/WSL/Linux)
- Same physical directory structure used for all platforms (on shared network-drives)
- Generated, full compliant 'make' files
  - 100% reliable builds
  - Cross reference
- Allows subdivision into SubSystems and Components
- Any number of SubSystems and/or Components
- Any number of libraries and/or executables per Component
- Strict applicable scoping rules
- Support for generation of 3rd party software
- Integrated support for any compiler
- Integrated support for Auditing tools like QAC, QAC++, PCLint and C++Test
- Integrated support for Documentation tools like Doxygen
- No user-written scripts
- Support for multi-site environments
- Command-line oriented with GUI
- Support for GUI integration (e.g. Visual Studio, SlickEdit, Eclipse)
- Automated directory creation and structure setup
- Independent from Configuration Management System (CMS)
  - CMSs supported (for automated structure creation), SubVersioN and Git
- Parallel generation (also in ‘grid’)
- Background generation ('at' jobs) with extensive logfile
- Prepared for tools like 'Softfab', ‘BuildForge’, 'Hudson' and 'CruiseControl'
- Uniform way of working
- Simple in use. Easy to learn. **Powerful** due to simplicity and consistency
- Suitable for small, medium and large systems
- Only dependent on Perl (Version 5.16 or later)

Final Remarks & Questions

- No 100% solution
- Combat proven
- Requires change of focus

- Questions
  - Task Lifecycle
  - Problem Areas
    - Directory Structure
    - Scripting
    - Integrity
    - Variants
    - Tooling
  - The Solution
    - Flat and rigid directory structure
    - make-file generation
    - The Bigger Picture
  - Does it Work

Final Remarks & Questions

Smart people find complex solutions

Intelligent people find simple solutions