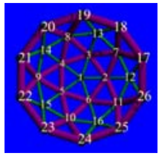
Buckminster | Component Assembly Project



A subproject of the Eclipse Technology Project

Problems?

GrowingPains Inc. – Starting out

Startup – financing completed!

- Team starts coding
  - Code base is small
  - All know whole system
  - Building is easy & fast
  - Communication is easy & ad hoc
  - Only one configuration

High pressure to reach the market in time!

GrowingPains Inc. – Leveling out

Success – v1 out the door after 6 months!

- Team expands to build v2 and maintain v1
  - Code base grows
  - Harder to know whole system
  - Building is somewhat complex and buildtimes go up
  - Communication must be more formal and less ad hoc
  - # of supported cfgs/variations increases

High pressure to timely fulfill market expectations!

GrowingPains Inc. – Strain starts to tell

Success – v2 out the door after another 18 months!

- Team expands and becomes physically distributed to maintain v1/v2, and laterally expand the market to new cfgs/variations based on v2
  - Code base grows really big
  - Product portfolio expands
  - Impossible to know whole system
  - Building is a black art; full builds take looong
  - Communication is hard, esp. given time zone diff:s
  - # of supported cfgs/variations increases exponentially
  - New hires have a hard time becoming productive
  - Many unnecessary 'non-problems'

High pressure to follow up market success!

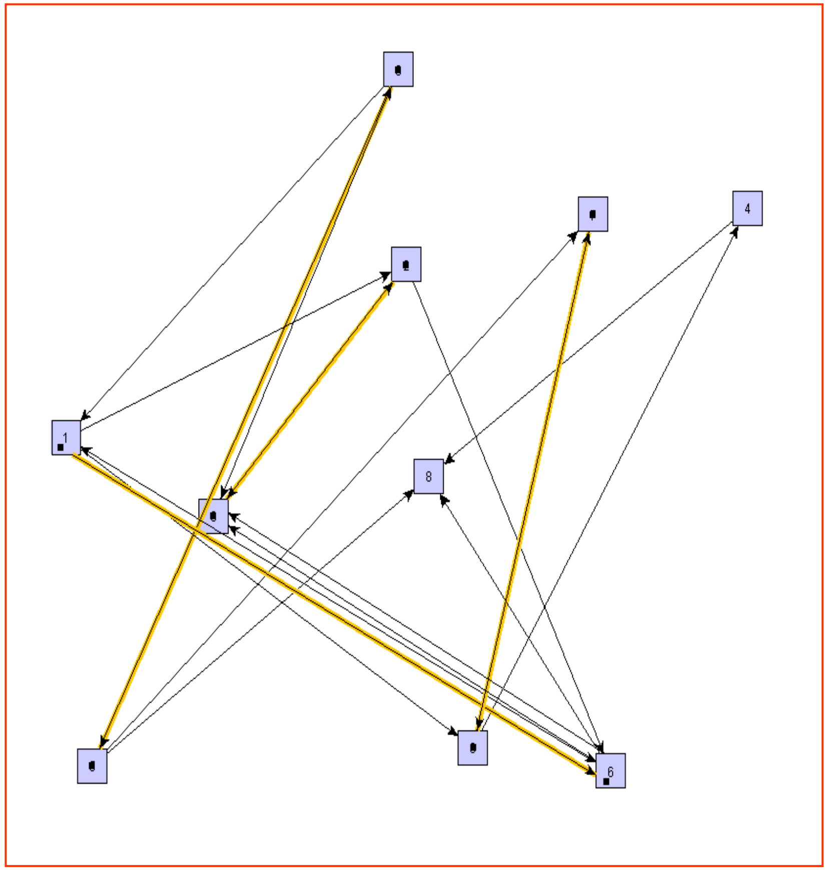
GrowingPains Inc. - Victims of success?

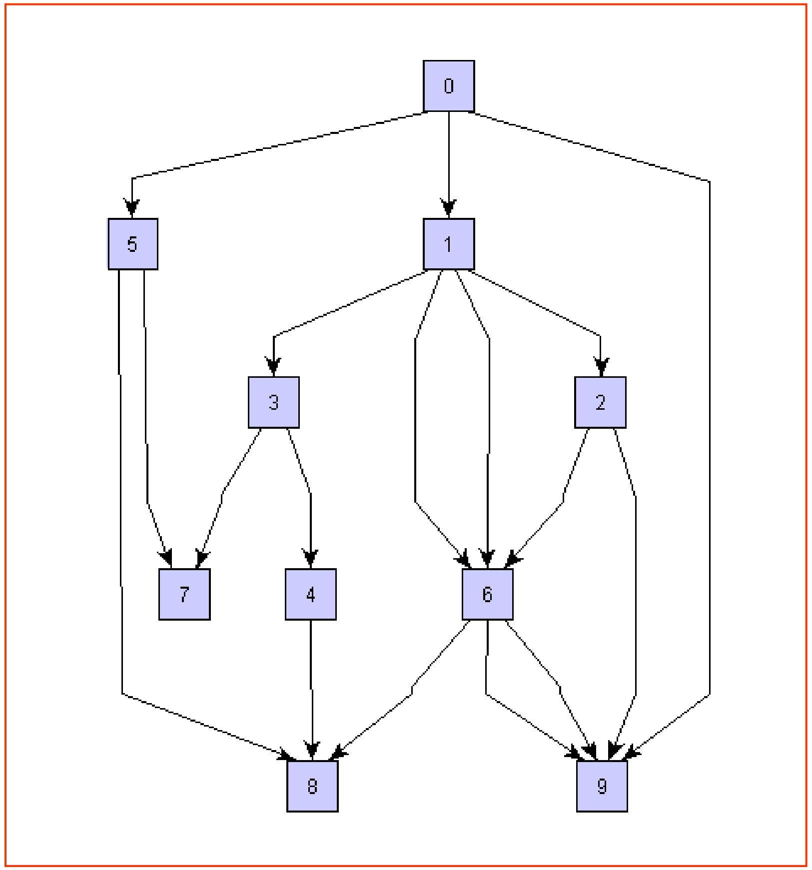
The situation is understandable, but really requires addressing

So, what’s to be done?

Many issues, so presently we’ll focus around monolith codebase related issues

Straightening out the kinks





The componentized monolith

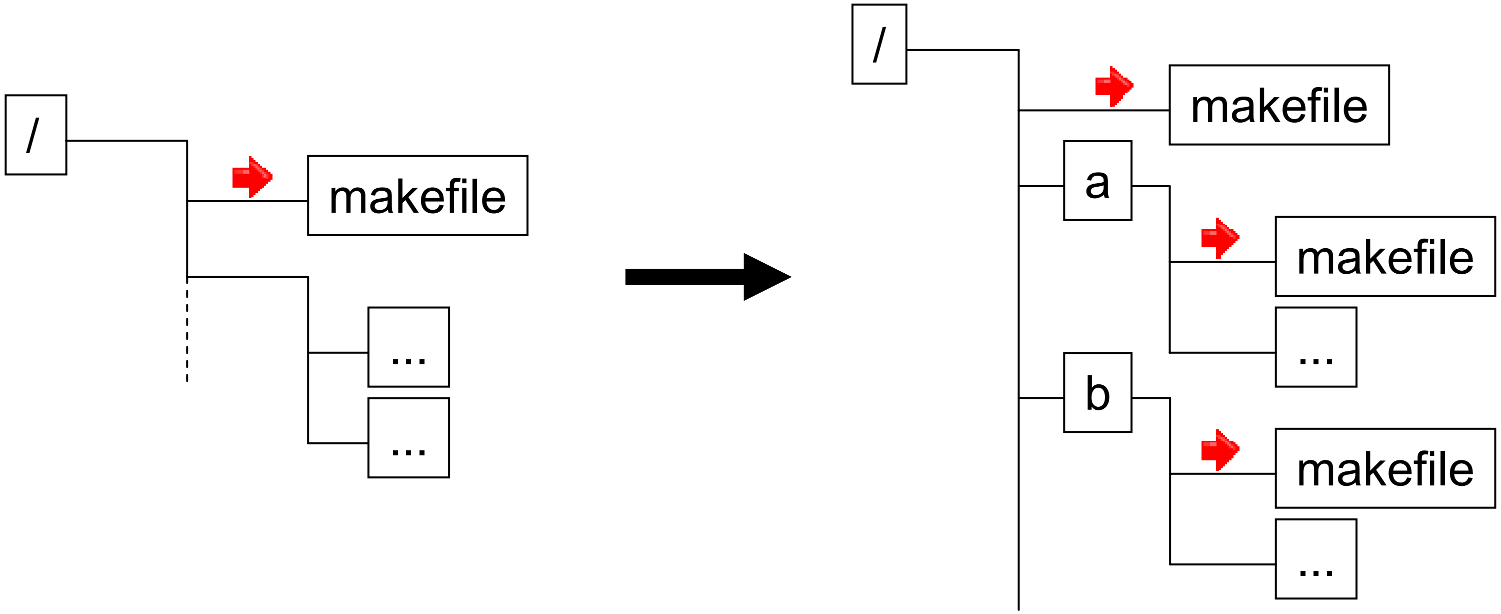
- A new & better monolith can be arrived at
  - It is generally a necessary first step
  - But, will only scale a finite amount better
- Regardless, there are some definite benefits:
  - Clearer separation of concern
  - Responsibility can be disseminated among persons/groups
  - Software architecture can be made clearer
    - For example, by separating external from internal API

Component hierarchy becomes an important topic

- For example, no circular dependencies allowed!
  - Impossible to calculate build order
  - Java allows it because the compiler handles it, but you can’t create regular makefiles for it. May be acceptable **internally** for a component though.
- A monolith lacks some flexibility
  - What to do if a variant is needed where only a single subsystem is upgraded?
    - The general problem is clear: how to describe a configuration with both ’what components’ and ’what **versions** of those components’. Then, how are such configurations materialized to disk (from potentially several sources).

Sample scalability issue...

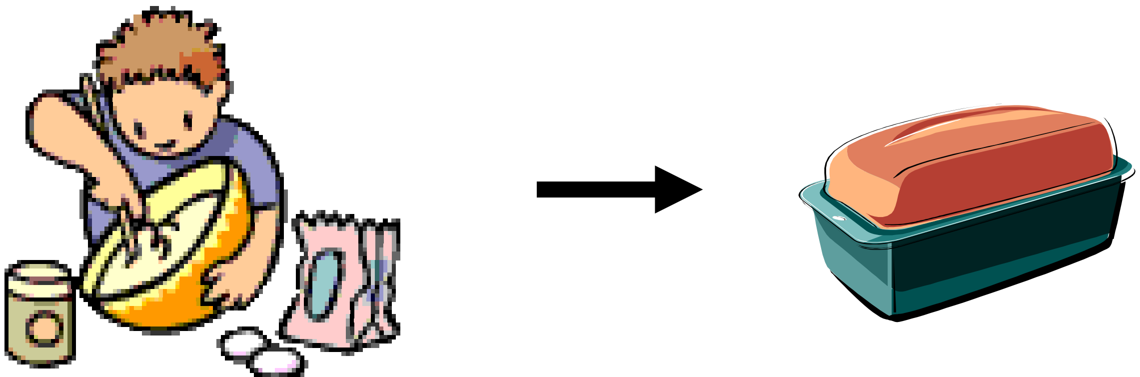
- Build times does not necessarily improve; in fact, it can get worse!
  - Builds are still made begin-to-end each time
  - Due to otherwise improved structuring, a componentized build script might perform worse*



* "Recursive Make Considered Harmful" (http://aegis.sourceforge.net/auug97.pdf)

Common solution:
Replace source with prebuilt libraries

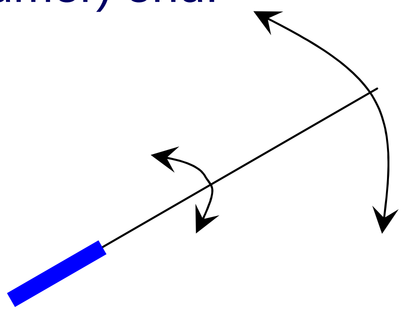
- Good solution, but generally has some weaknesses
  - requires management and tool support to ensure correctness
  - is difficult when you need to run with the source for debugging purposes (what source corresponds to what binary, what changes are needed in make files to dynamically adjust to etc)



- A tool solution needs to be able to dynamically use rules to switch between one form or another, often according to user intentions, and transparently propagate knowledge between otherwise ignorant components

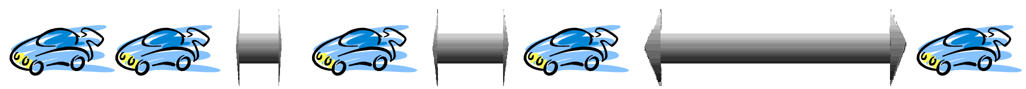
The consumer/producer problem

- Small/smooth movements at the (producer) base translate to wild/jerky motion at the (consumer) end!



- ...compare to a whip:

- ...or a car convoy:



So, producers are frequently driving the rate of change

- Which means consumers:
  - has a hard time to influence timing
  - frequently has to work in an unstable environment

Consumers may not be able to control the rate of change...but they should have control of accepting changes at their own rate. More power to the consumer!

Separation achieved but needs to be maintained

- Beware: developer inventiveness and time pressure can quickly erode componentization
- Cheats can go undetected a long time (and, you can be sure they will rear their heads at the most inopportune moment...)

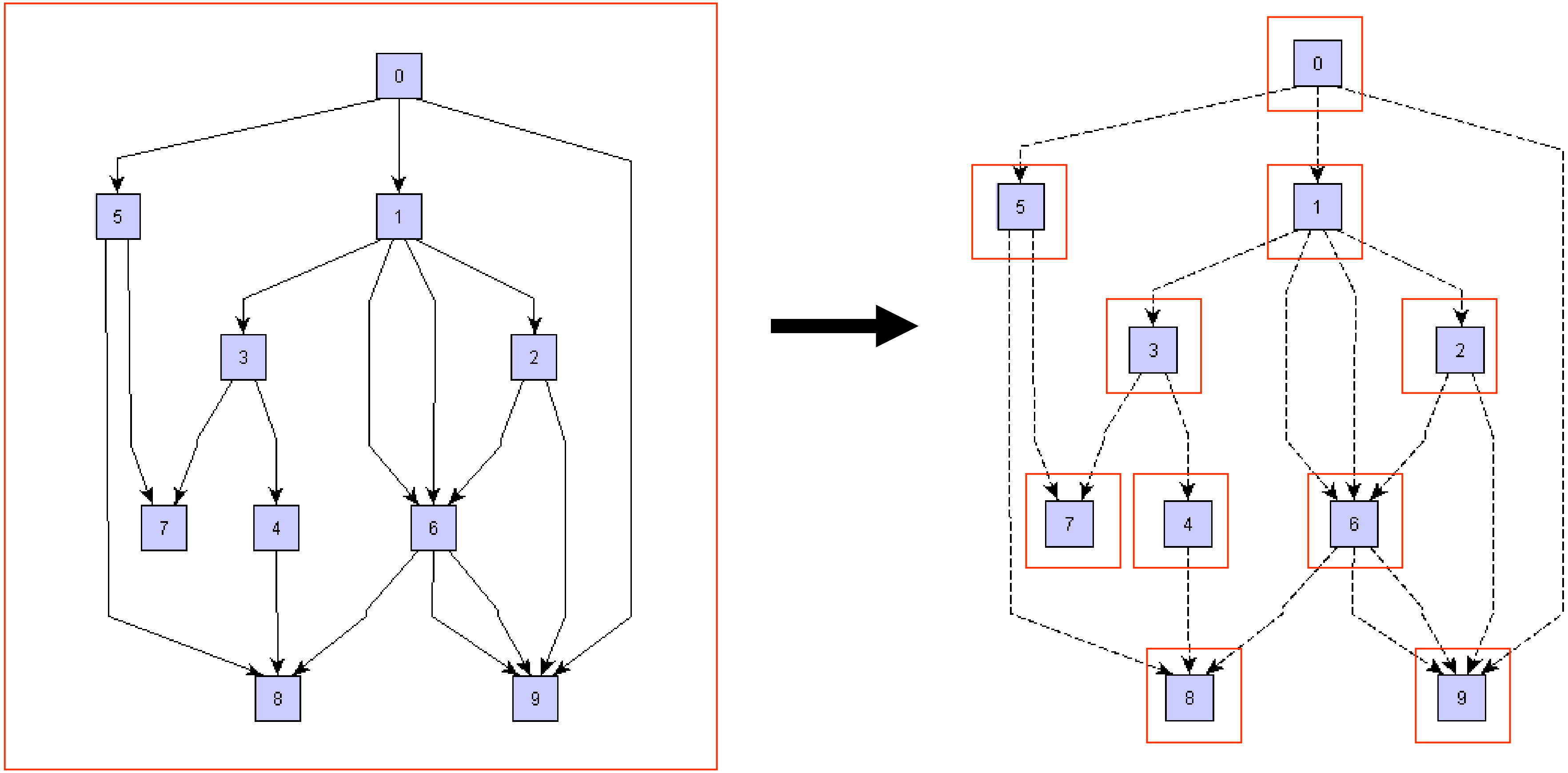
Actually, it’s sometimes ok to cheat, but it should be:

- Harder to do casually
- More visible
- More localized

Componentizing is good, but a monolith is still a monolith...

Next step!

Divide & Conquer!



Increase reusability opportunities!

- Apply OO principles on several levels
- Data hiding, encapsulation, abstractness
  - Encourages separation of concern
  - Encourages interface & implementation splits
  - Encourages looser coupling
  - Encourages better design mapping
- Explicit component dependencies more visible
  - Helps with impact analysis
- Enables higher parallelism
- Clearer areas of responsibility

Component FOO
Depends on A,B,C
Public bin layout
Private src layout

Some assembly required, though

- New/adapted tools & processes – sample issues:
  - Components have individual life cycles (but pay attention to what co-varies and avoid incorrect componentization)
  - Impacts how Configuration Management/Version Control is performed
  - Keep track of configurations (components/versions, dependencies)
  - A generic build initiation framework understanding piece-by-piece builds, combine builds, persisted intermediate results etc
- Systems & people must ’talk’ using the same terminology
- A fair amount of admin/management/development/maintenance
- Must be open enough to wrap an existing environment before progress can be made

Solution?

Buckminster overview

Summary

- The Buckminster high-level definition statement:

  Buckminster addresses development problems associated with assembling complex component structures in team-based development

- Buckminster makes use of, and leverages, Eclipse and its architecture/framework. Works as both a fully integrated UI in the IDE, but also as a freestanding (command-line) executable environment.
- Project name: after Buckminster Fuller, architect, engineer; inventor of the geodesic dome; pioneer of manufactured modular structures.

> *"When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."*
>
> **R. Buckminster Fuller**
> *US architect & engineer (1895 - 1983)*

Key objectives

- Buckminster’s primary objective is to leverage & extend Eclipse to:
  - bring complex component development on par with current mechanisms for plug-in & feature development
  - extend the component dependency model to allow materialization based on match rules
- Buckminster will accomplish this by:
  - introducing a project/component-agnostic way of describing arbitrarily complex component structures and dependencies in development projects
  - allowing component materialization based on match rules, i.e.similar to platform mechanism for runtime resolution of plug-ins/features
  - providing a materialization mechanism that handles all component types referenced through repository handlers

Features

- Buckminster currently includes:
  - Complex dependency resolution
  - Uniform component dependency formats
  - Intelligent retrieval mechanisms
- Buckminster is itself componentized and has several possibilities of being extended easily (through the generic Eclipse ’extension point’ mechanism).

Drill-down: Complex dependency resolution

- Provides recursive resolution of dependencies
- Supports a variety of versioning schemes
- Applicable to source and binary artifacts that are not version-controlled in a traditional sense
- Uses match rules similar to those in the Eclipse plug-in runtime framework, eventually allowing comparison of current and prior dependency resolutions to support update impact analyses

Drill-down: Uniform component dependency format

- Component-type agnostic mechanism for describing components and their respective targets and dependency requirements
- Will leverage dependency info associated with typical Eclipse projects and range of other component types
- Extensible to provide additional strategies for dependency pattern recognition

Drill-down: Intelligent retrieval mechanisms

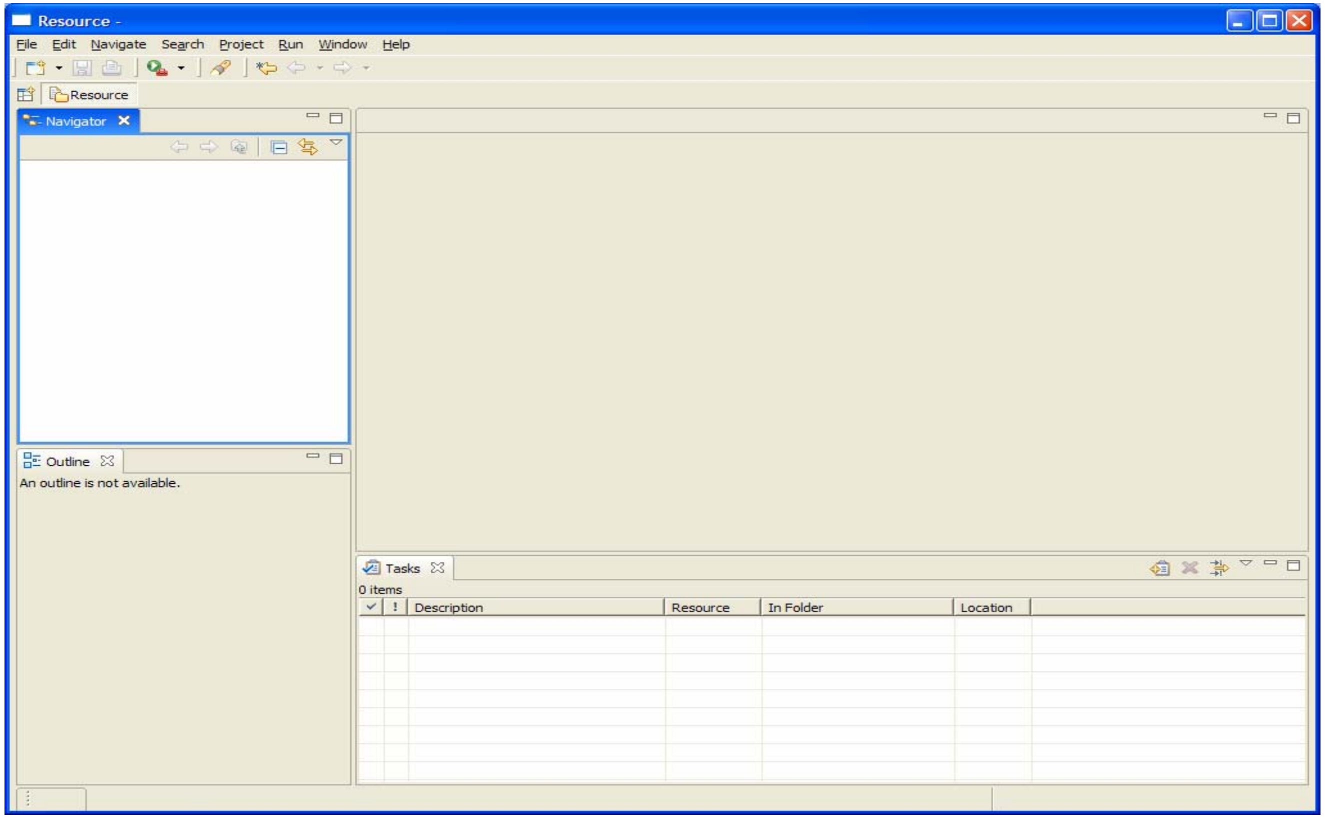
- Separating the bill of material needed for a given configuration from its actual materialization
- Separation is of value since:
  - dependencies may appoint software that is locally installed on one machine but lacking on another
  - bills of materials may be shared between team members, while materialization info may vary
  - information about repositories will be abstracted out in order to provide site and repository transparency

Demos

- Buckminster in action

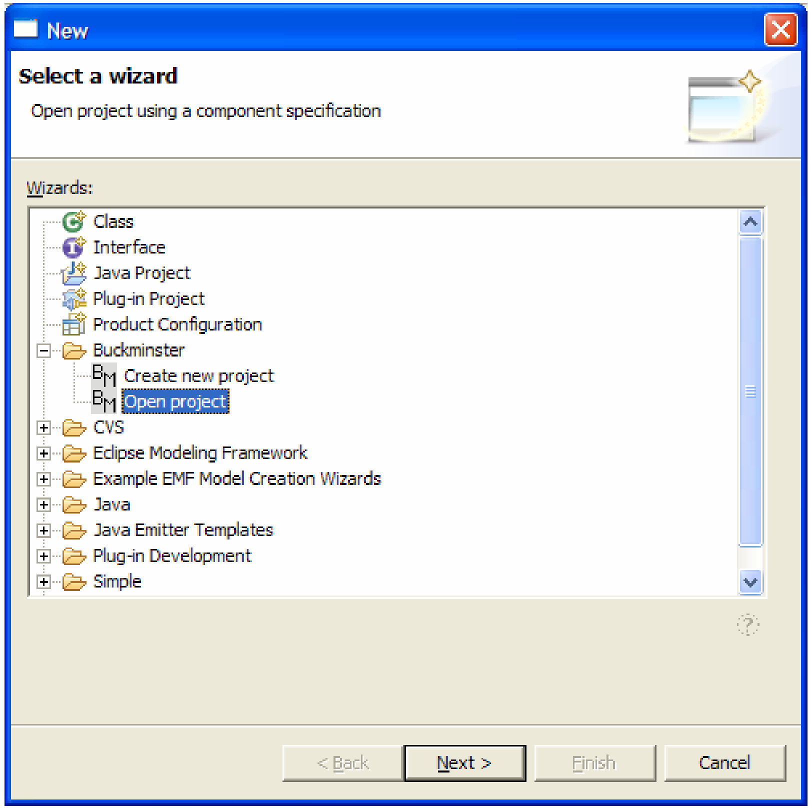
Buckminster in action – sample scenario (1)

- Scenario: ’I want to write code for Buckminster’
  - Fire up Eclipse



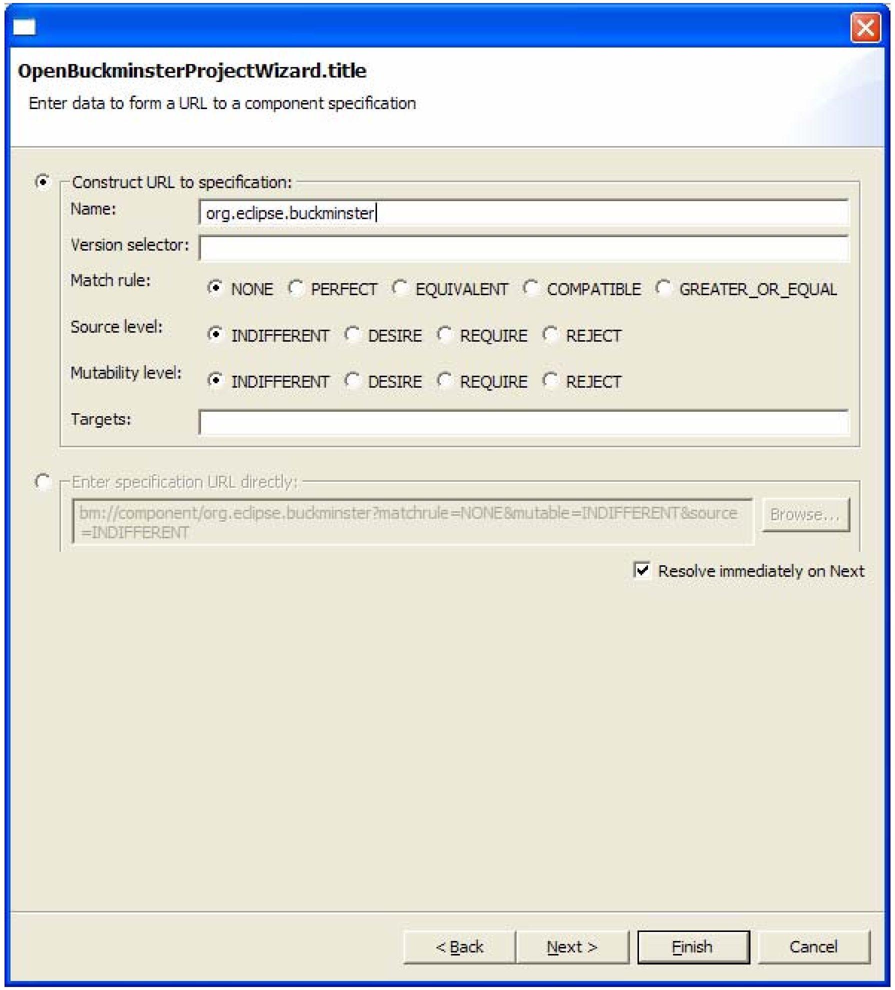
Buckminster in action – sample scenario (2)

- Use the 'Open Project' Wizard



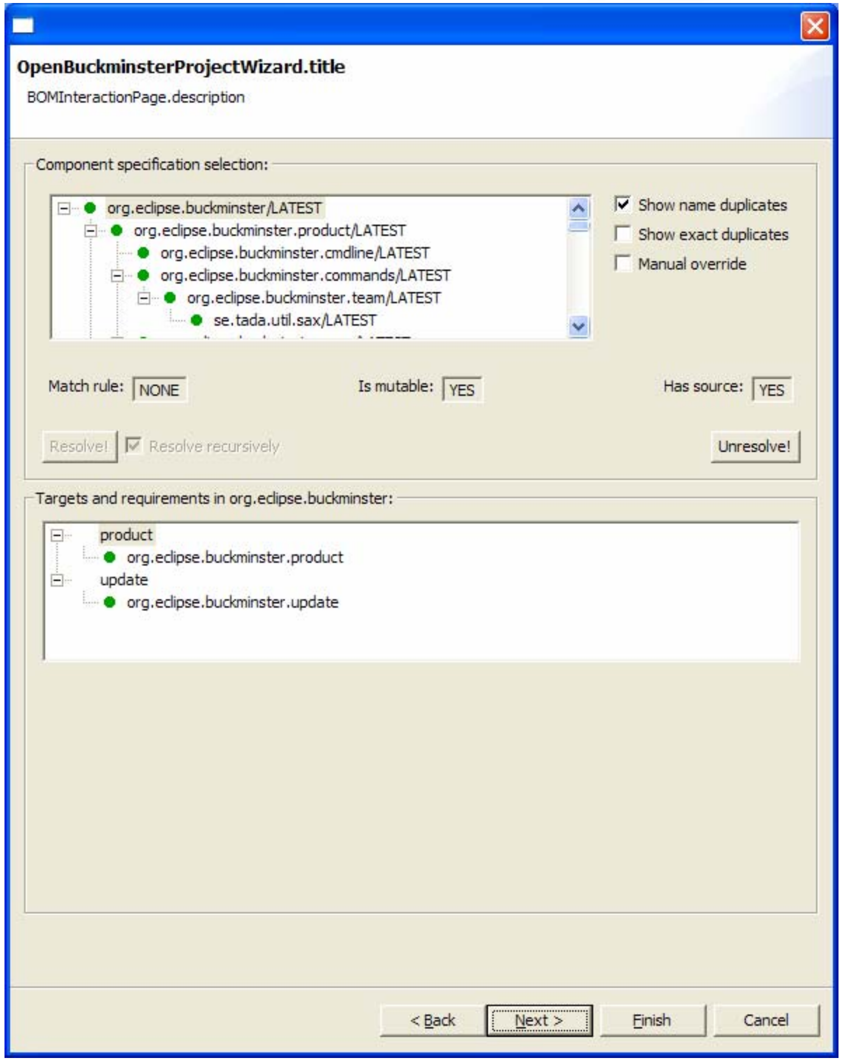
Buckminster in action – sample scenario (3)

- Enter the ’known’ entrypoint
  - org.eclipse.buckminster
- With the given component name, the resolver can figure out in what repository it should look.
  - Either press Finish outright, or use Next...



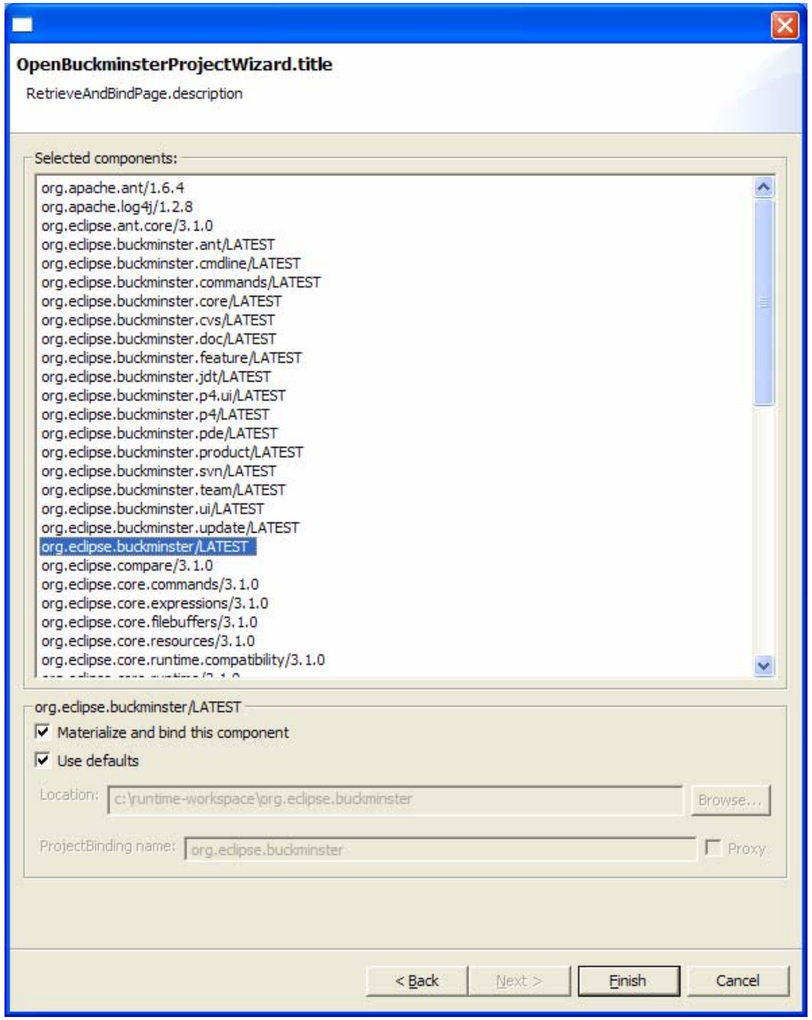
Buckminster in action – sample scenario (4)

- The wizard has walked the entire depency tree (in this case about 60 components required) and made default selections.
  - Each component can be resolved 'by hand' if desired
  - Many components are fulfilled by the running Eclipse instance



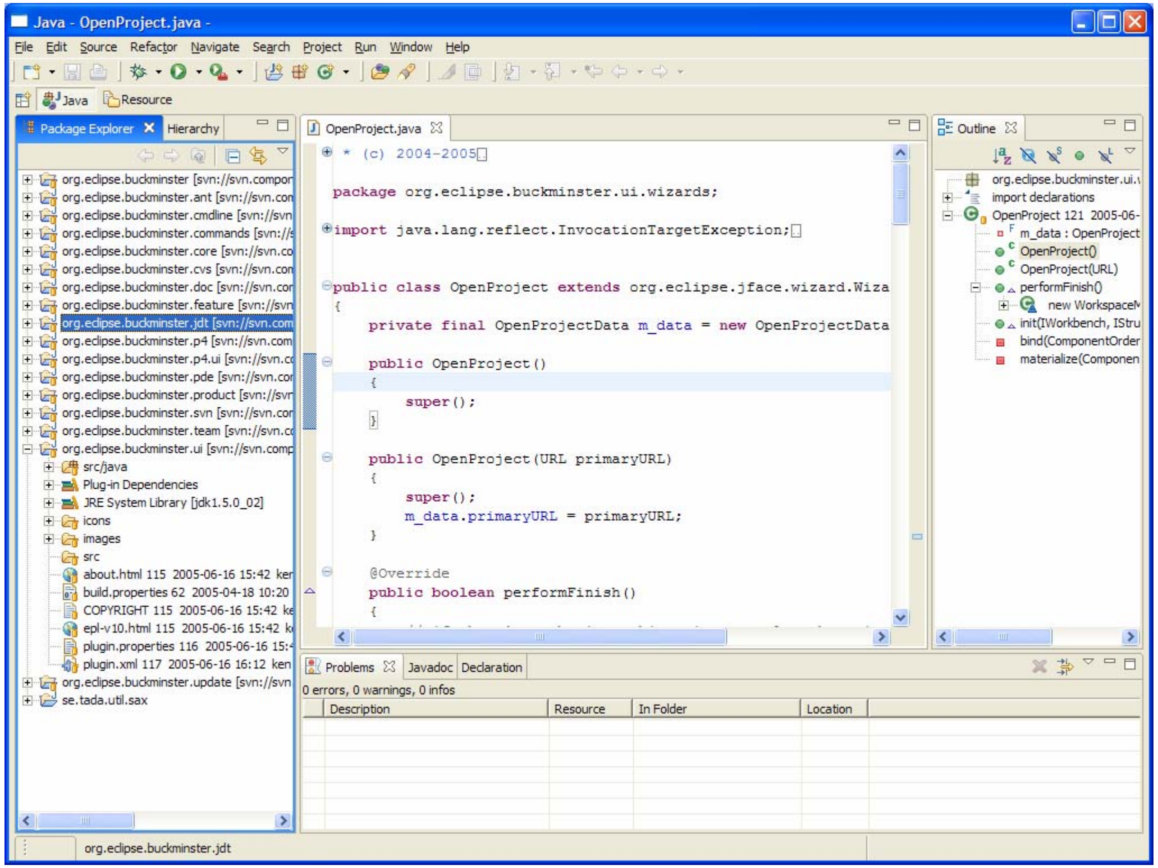
Buckminster in action – sample scenario (5)

- The final step is to 'materialize and bind'
  - I.e. download code from respective repository and make it visible to Eclipse as native 'projects'
  - Again, most components are fulfilled by the running Eclipse instance itself, thus no materialize/bind required



Buckminster in action – sample scenario (6)

- The required components are now ready for use



Thank You

Please visit:
http://www.eclipse.org/buckminster