Product Line Evolution

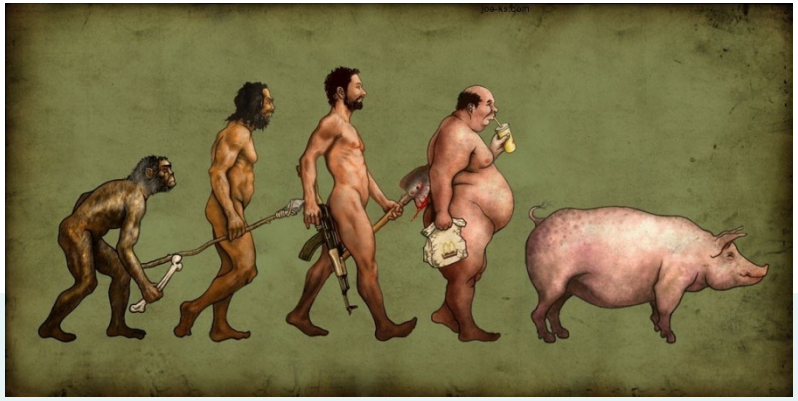
KV Product Line Engineering (343.354)
Dr. Roberto Lopez-Herrejon
Dr. Rick Rabiser

25.5.2016

Contents

- What is software evolution?
  - Basics of software evolution: why, how, when, where, who, what …?
- Evolution of Product Lines
  - Why is evolution of product lines more complex?
- Component-based vs. Model-based evolution
  - Example: Koala and DOPLER

If software were a man this is what software evolution would look like



What is software evolution?

- “Evolution is what happens while you’re busy making other plans.”

- Usually, we consider evolution to begin once the first version has been delivered:
  - Maintenance is the planned set of tasks to address changes
  - Evolution is what actually happens to the software

Evolution is important

- Organizations made huge investments in their software systems - they are critical business assets
- To maintain the value of these assets for business, they must be changed and updated
- The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software

Software change is inevitable

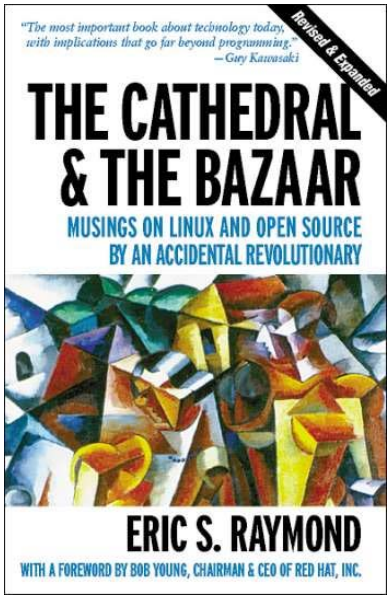
- New requirements emerge when software is used
- The business environment changes
- Errors must be repaired
- New computers and equipment are added
- The performance or reliability of the system may have to be improved
- etc.

A key problem for organizations is implementing and managing change to their existing software systems

Evolution Styles: Cathedral and Bazar

JOHANNES KEPLER UNIVERSITY LINZ

- Cathedral
  - careful control and management
  - debugging done before committing code
  - evolution is slow, planned, rarely undone



- Bazaar (Open Source SW Developmt)
  - lots of low-level changes, frequent fixes
  - lots of “building around” rather than wholesale changing, occasional redesigns
  - However: creeping “feature-itis”

Lehman‘s Laws of SW Evolution

JOHANNES KEPLER UNIVERSITY LINZ

Law	*Description*
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Maintenance Types

- **Corrective** maintenance: Fixing faults that cause the system to fail
- **Adaptive** maintenance: Accommodate a changing environment
- **Enhancement**: Adding new features
- **Perfective** maintenance: Improvements without effects on the end-user
  - make it easier to extend and add new features in the future
  - E.g.: re-engineering; refactoring
- **Preventive** maintenance: Preventing failures
  - by fixing defects in advance of failures (kind of perfective maintenance)
  - Key examples: Y2K and Daylight Savings adjustments

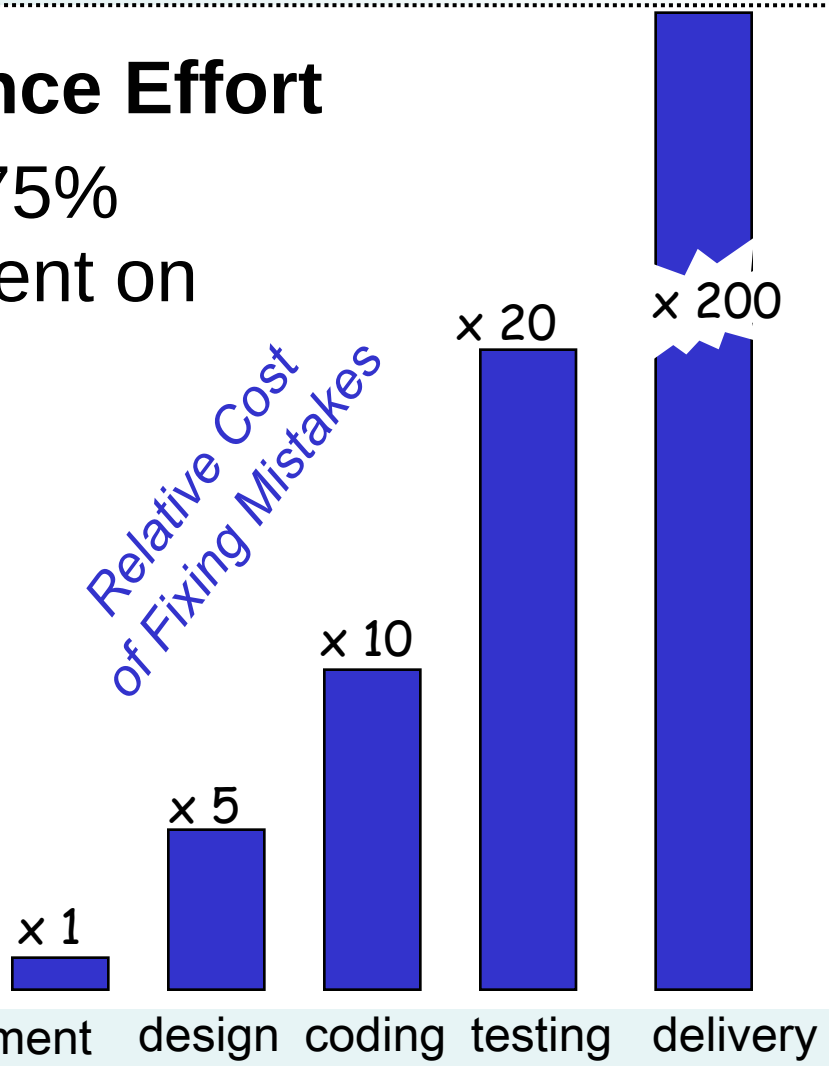
Maintenance Costs

- Usually greater than development costs (x2 to x100 depending on the application)
- Affected by both technical and non-technical factors
- Increase as software is maintained
  - Maintenance corrupts the software structure and makes further maintenance more difficult
- Ageing software can have high support costs (e.g., old languages, compilers, etc.)

Software Maintenance Costs

Relative Maintenance Effort

Between 50% and 75%
of global effort is spent on
maintenance



Maintenance Cost Factors

- Team stability
  - Reduced Maintenance costs if the same staff
- Contractual responsibility
  - No design for future change if developers have no responsibility for maintenance
- Staff skills
  - Maintenance staff inexperienced and limited domain knowledge
- Program age and structure
  - Structure is degraded and harder to understand and change

What about Legacy systems?

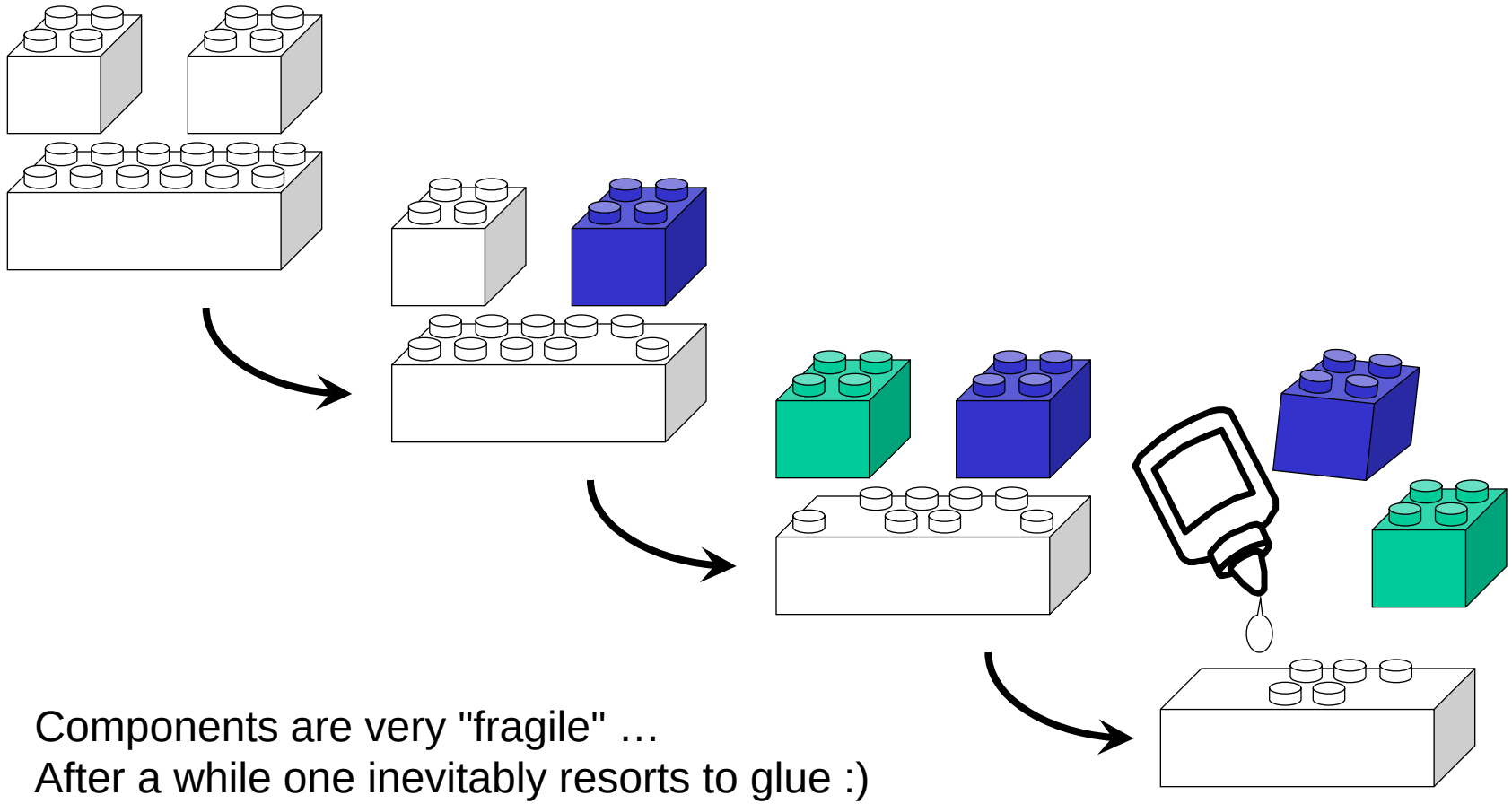
- Scrap the system/modify business processes so that legacy system is no longer required
- Continue maintaining the system
- Transform the system by re-engineering to improve its maintainability
- Replace the system with a new system

What about Objects?

OO techniques promise better flexibility, reusability, maintainability …

… but not for free! Remember our Duck example!

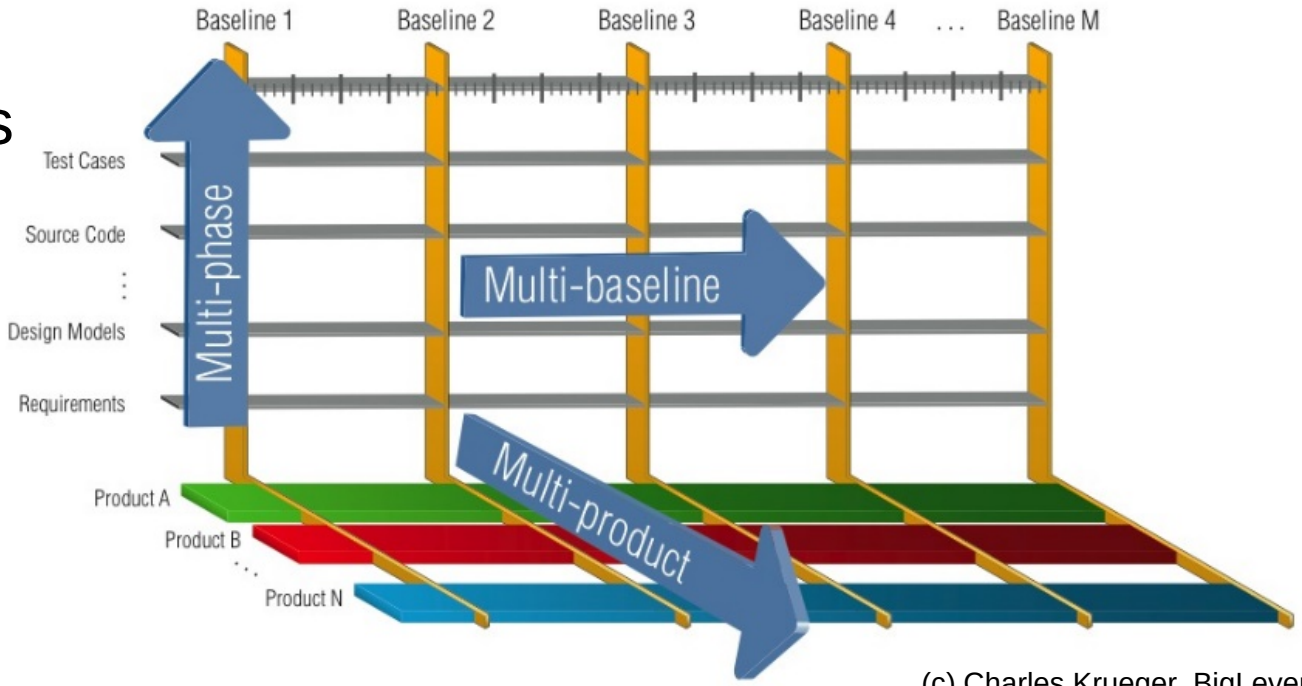
What about Components ?



Components are very "fragile" …
After a while one inevitably resorts to glue :)

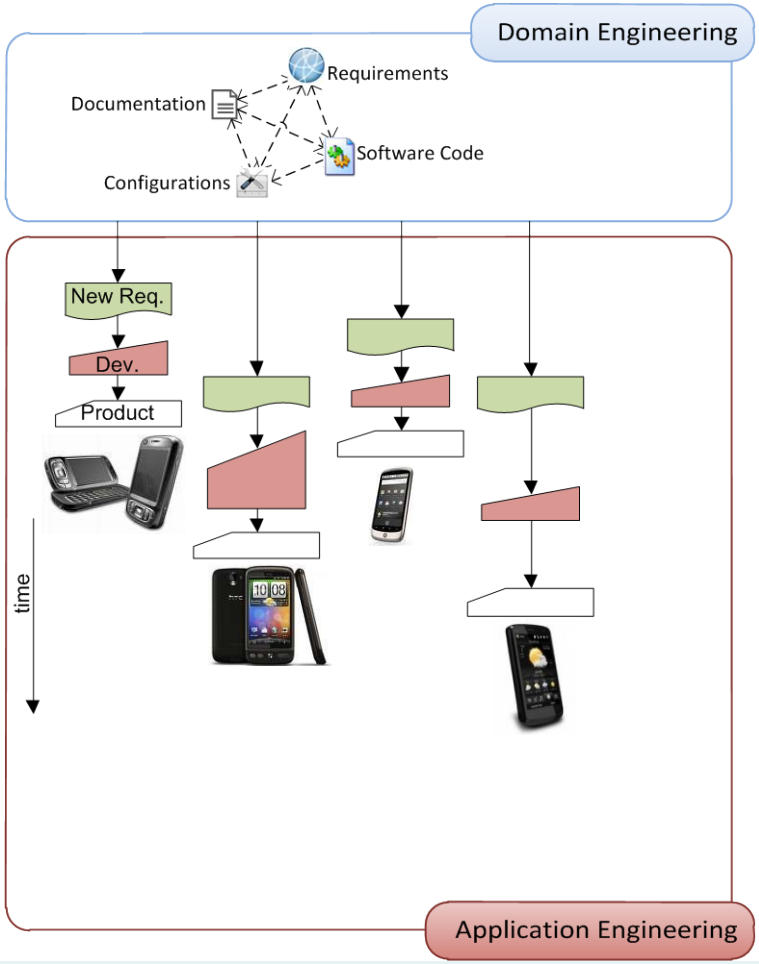
Is Product Line Evolution different?

- Longer life span?
- Added complexity: multi dimensional evolution
- Affects:
  - Core assets
  - Products
  - Features
  - …

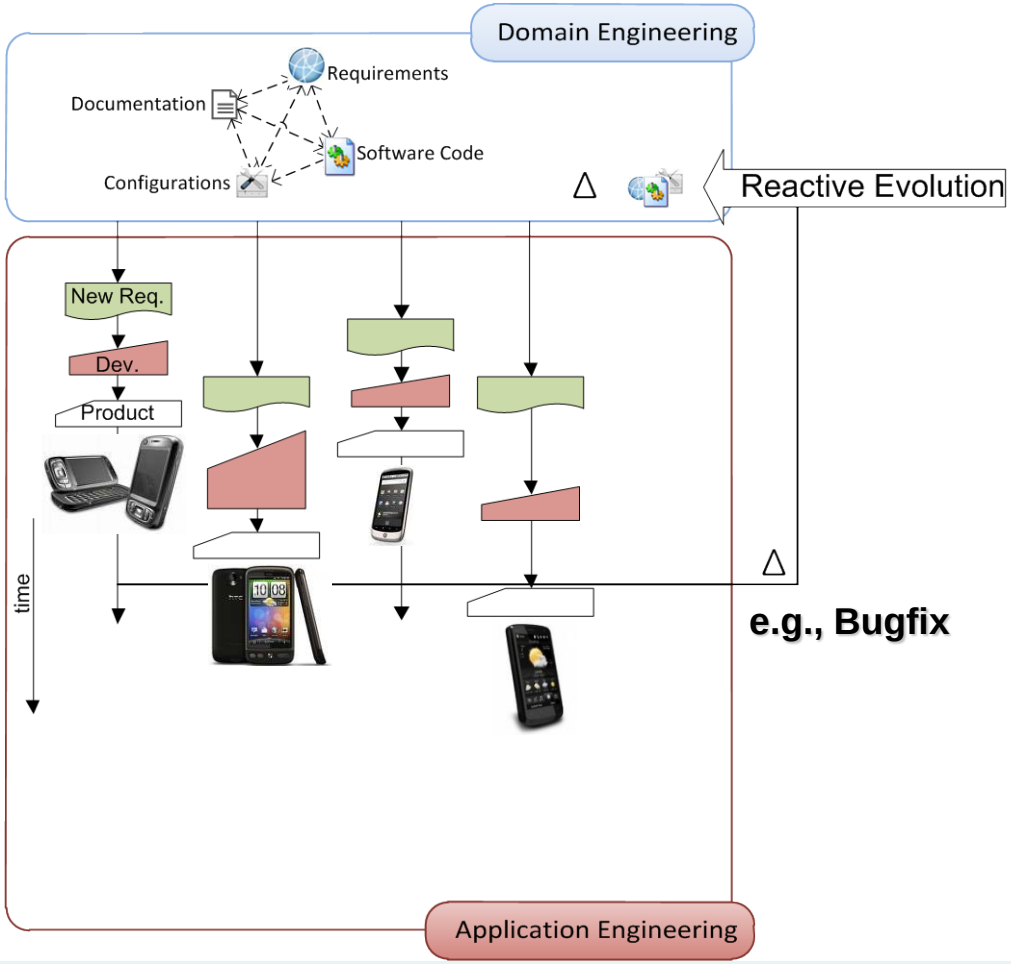


(c) Charles Krueger, BigLever

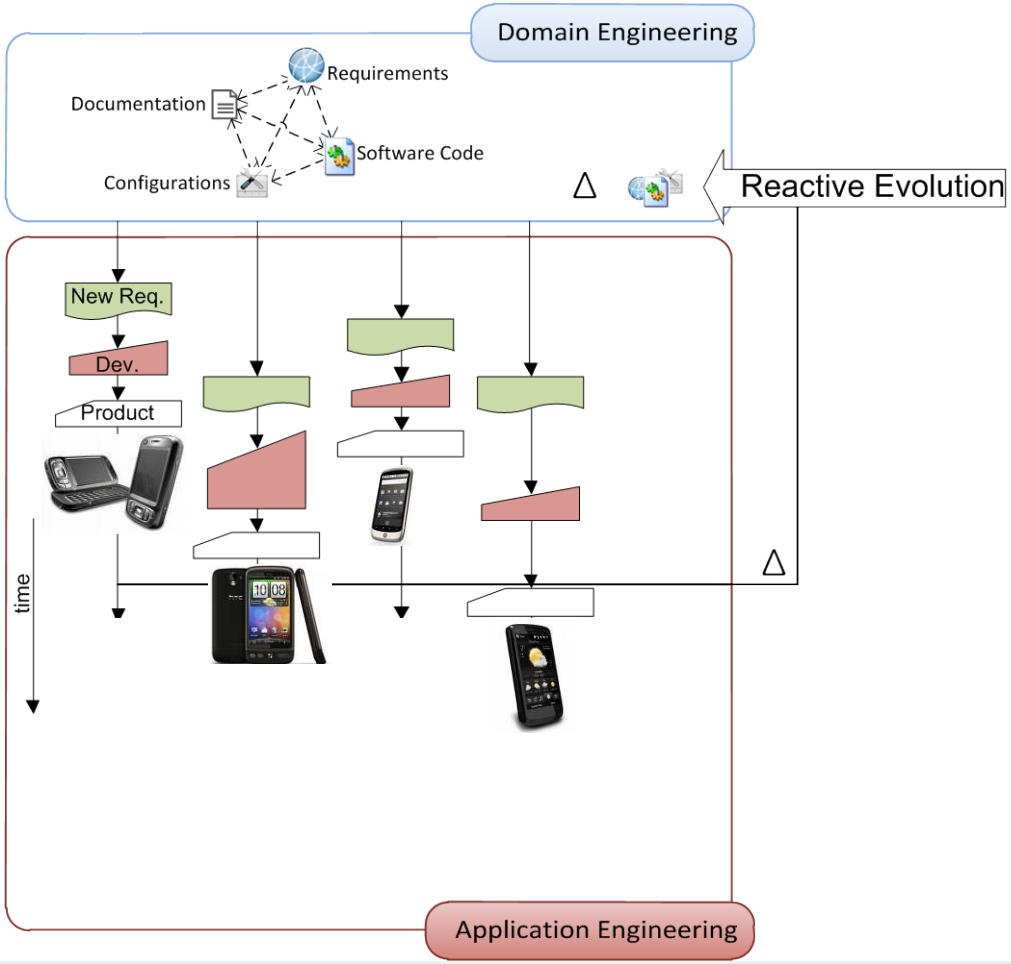
Sequential Product Line Engineering



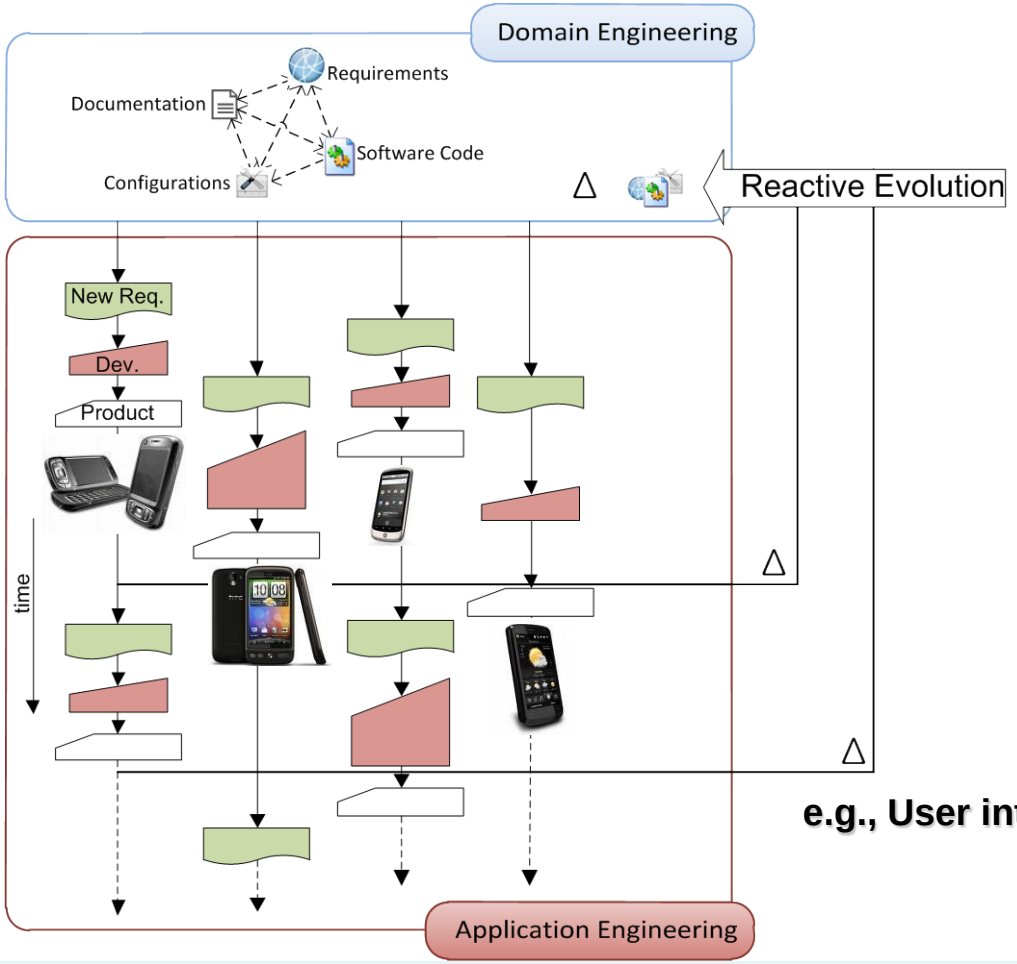
Sequential Product Line Engineering



Sequential Product Line Engineering

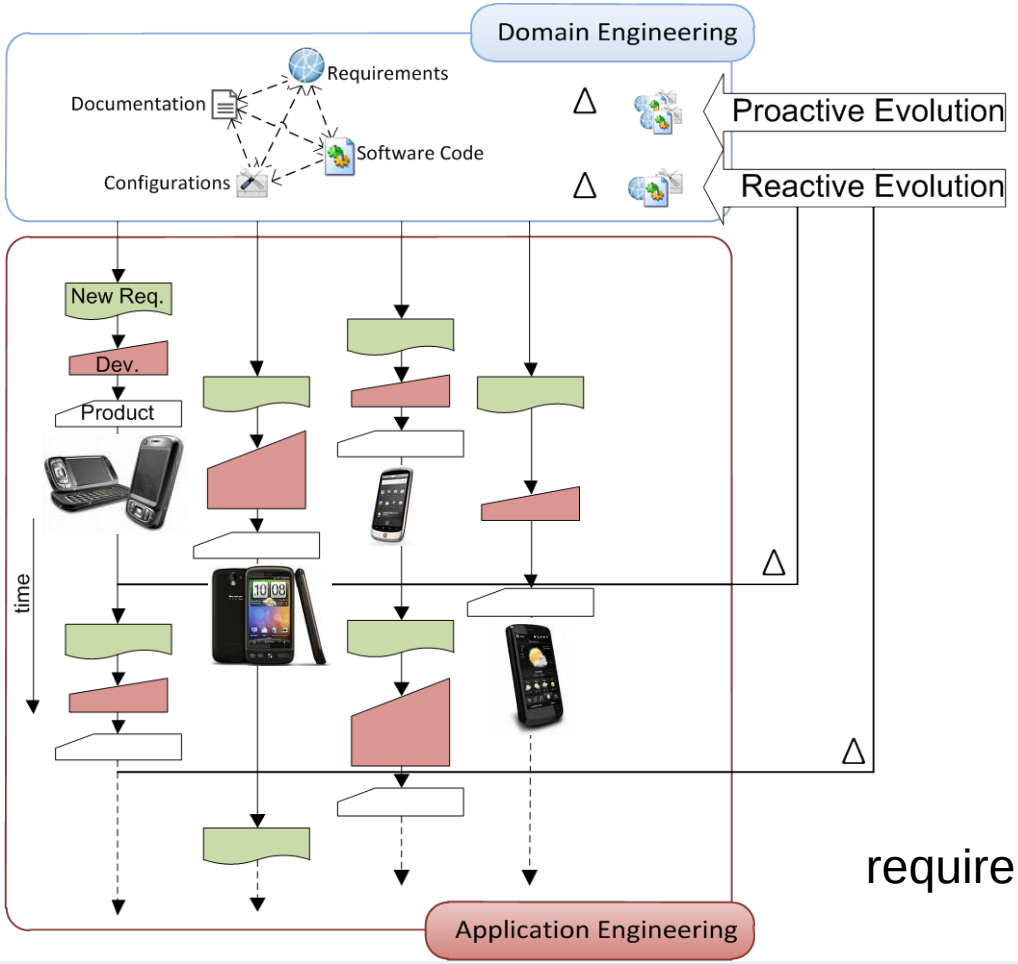


Sequential Product Line Engineering



e.g., User interface update

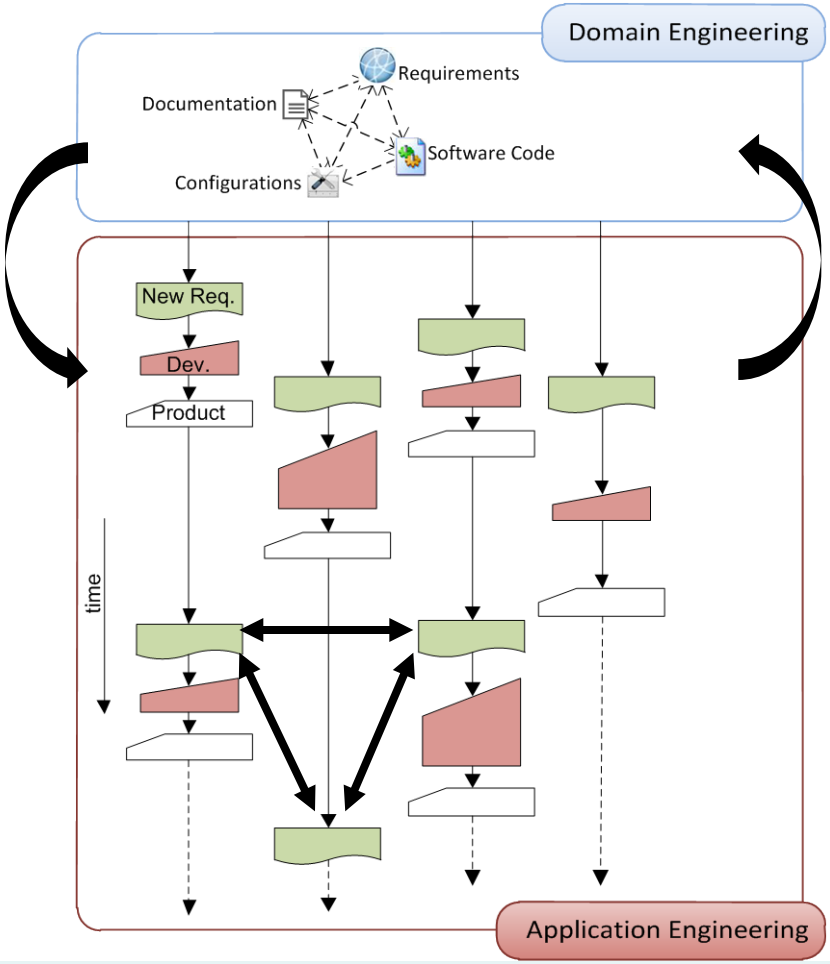
Sequential Product Line Engineering



e.g., Facebook integration

… consistency
… traceability &
requirements management

Sequential Product Line Engineering

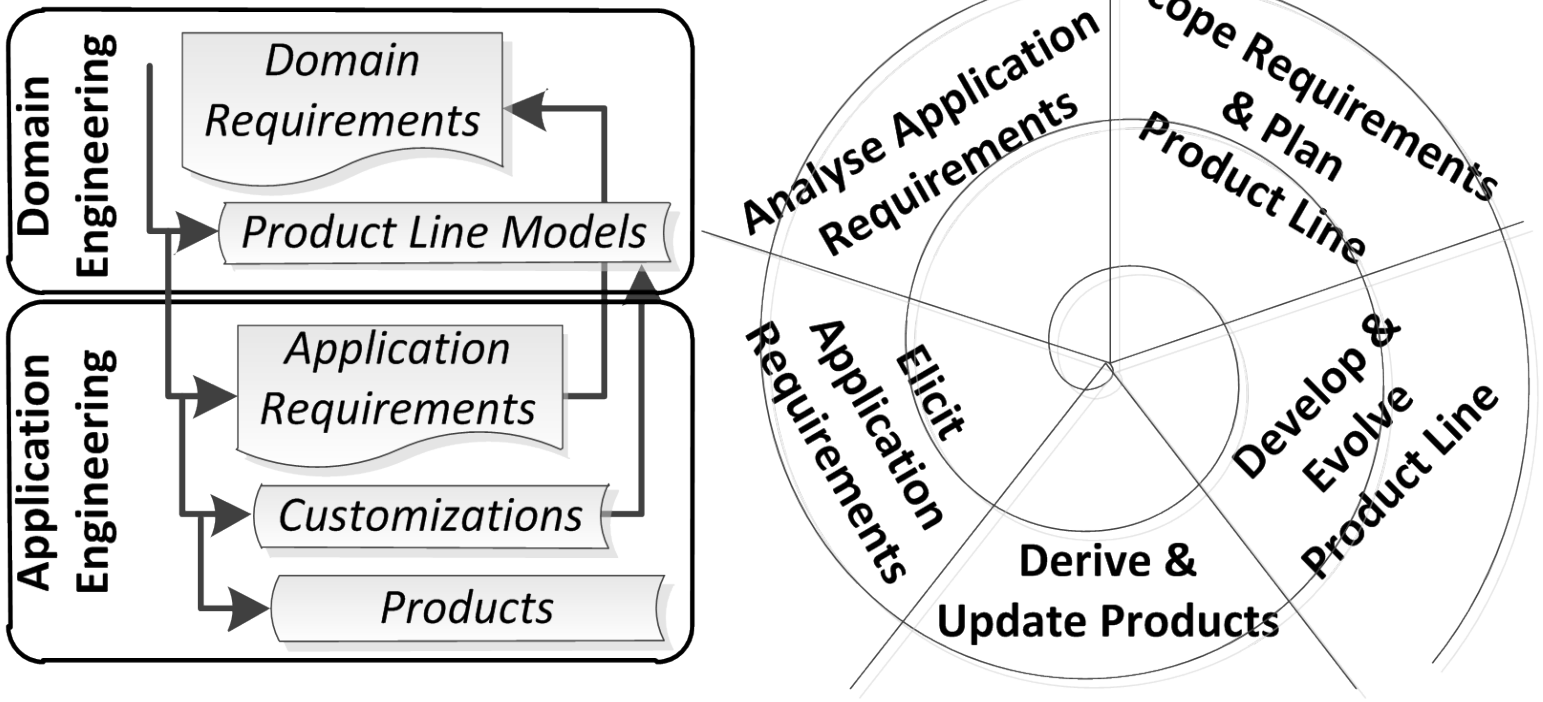


Issues

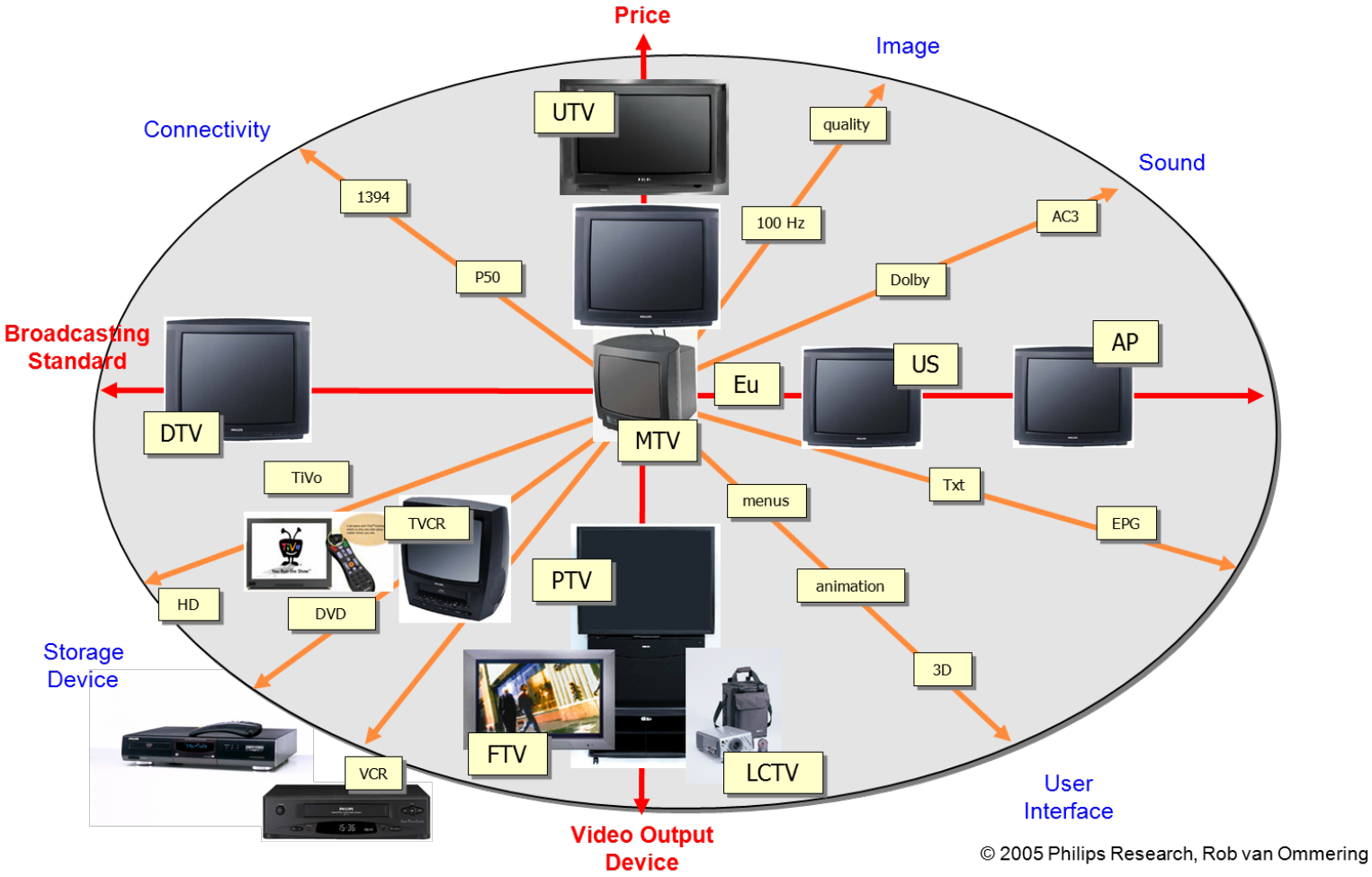
1. Interrelated requirements
2. Integration in Product Line
3. Integration in Products

→ Evolution cycle
round trip tool support

Sequential vs. Iterative PLE

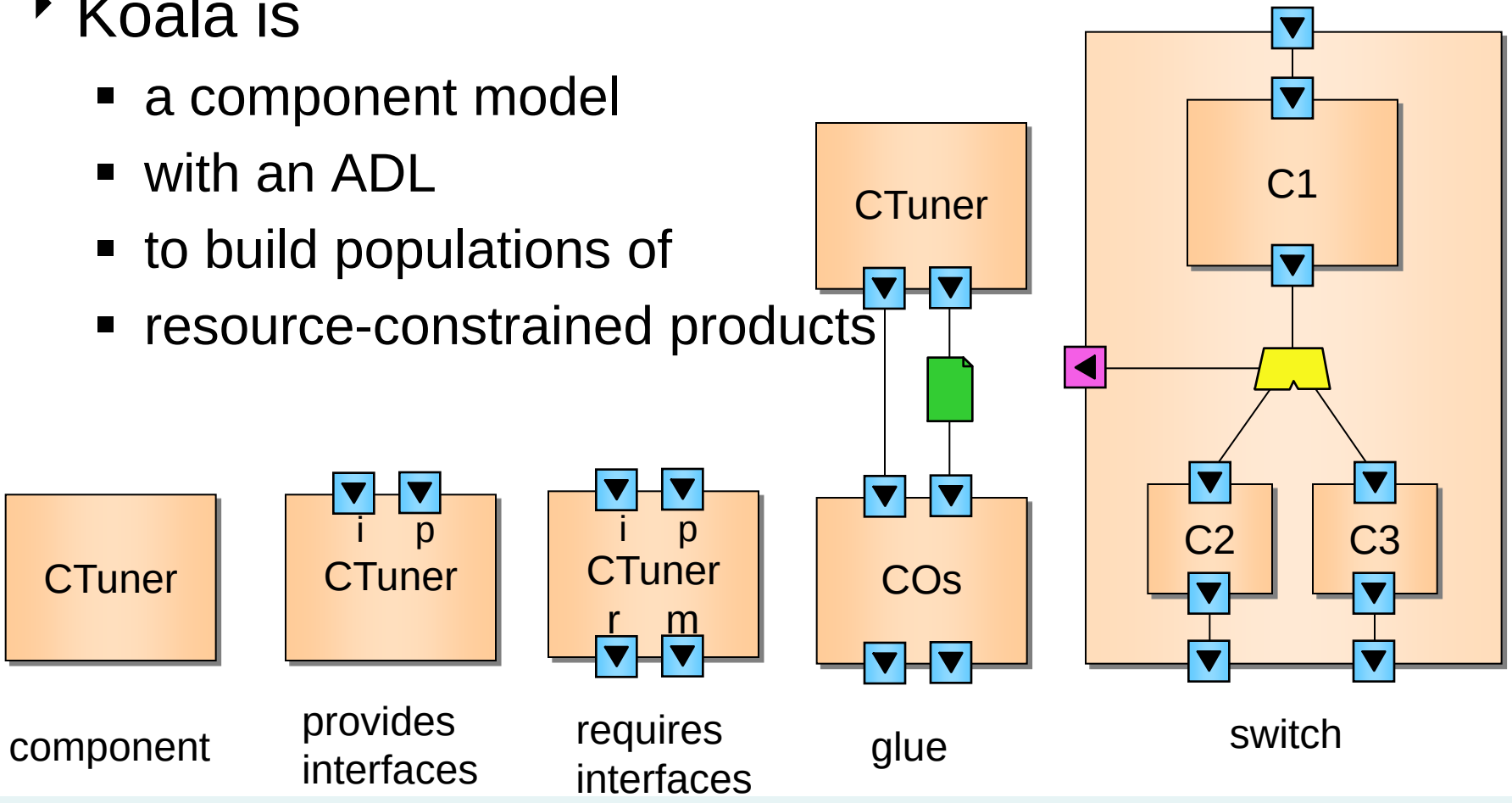


Example: Philips Product Line

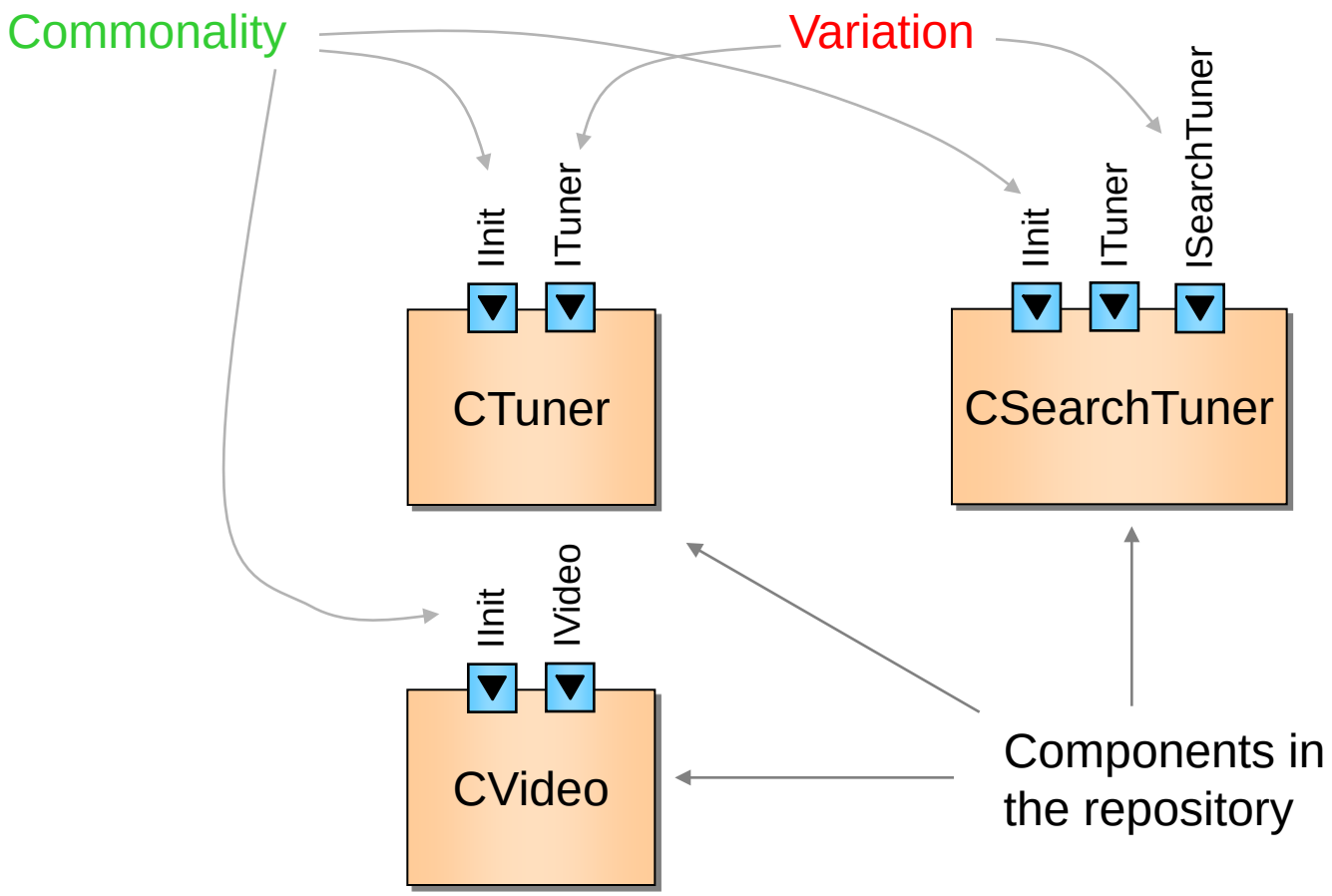


Koala Component Model

- Koala is
  - a component model
  - with an ADL
  - to build populations of
  - resource-constrained products



Commonality and Variation



Evolution of Koala Models

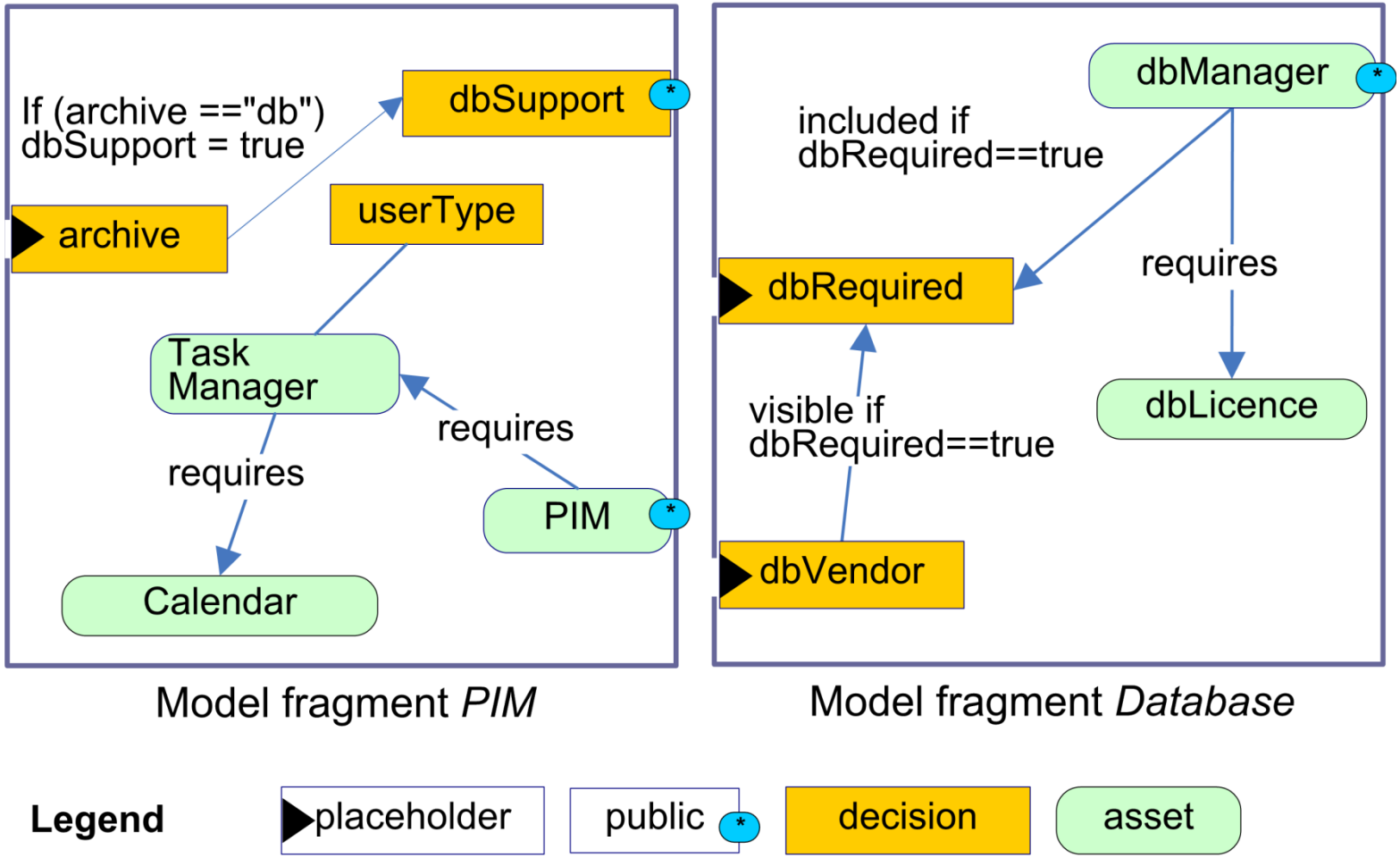
- Developers store interface definitions in a **global interface repository**
  - Existing interface types cannot be changed
  - New interface types can be added
  - An existing component can be given a new provides interface, but an existing provides interface cannot be deleted
  - An existing component can be given a new requires interface, but it must then be optional
  - An existing requires interface cannot be deleted, but it can be made optional

Our approach: (1) DOPLER Model Fragments

- Variability model fragments as units of evolution
- Each model fragment represents a stakeholder's perspective
- No detailed knowledge required about other fragments
- Model fragments are merged semi-automatically
- Merged model is not changed directly

- Product derivation requires merged model

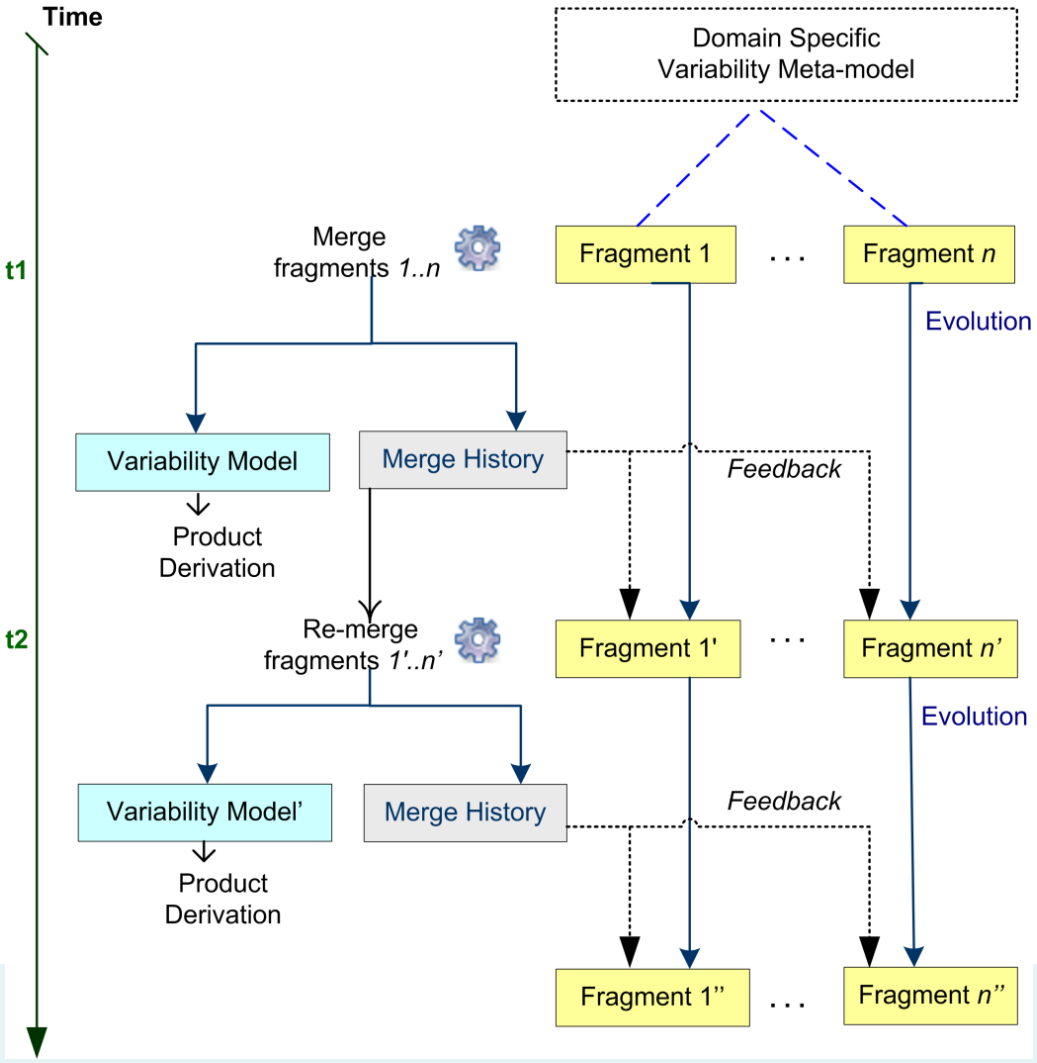
Example of Two Fragments



Semi-automatic merge process: conflicts, resolutions, and feedback

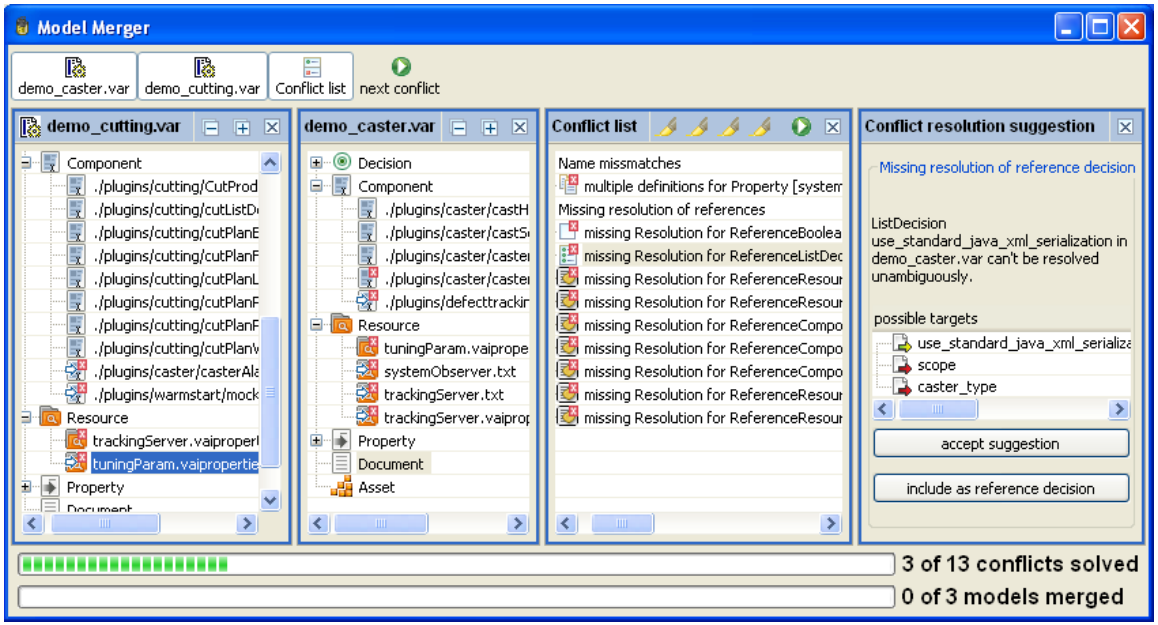
Merge conflict	Resolution strategy
Placeholder vs. element name mismatch	Synonym check with glossary Present to user *Feedback to fragment*: Rename one of the mismatching elements
Multiple definitions of elements	User can confirm semantic equality *Feedback to fragment*: Delete instances and use placeholders instead
...	...

Supporting variability model evolution

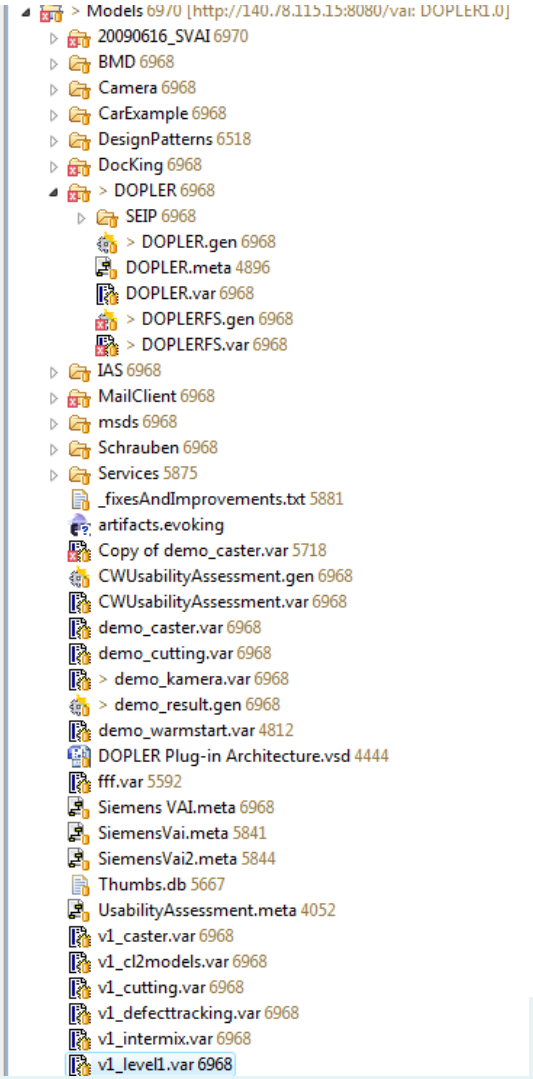


Fragment Merging Tool

- Semi-automatic merging of multiple fragments
- Relies on human intervention to resolve certain ambiguities



Our approach:
(2) Evolution Tracking in DOPLER



Software Product Lines are…

- Complex
- Maintained over many years
- Used in many different projects
- Typically built up with numerous interrelated modeling elements

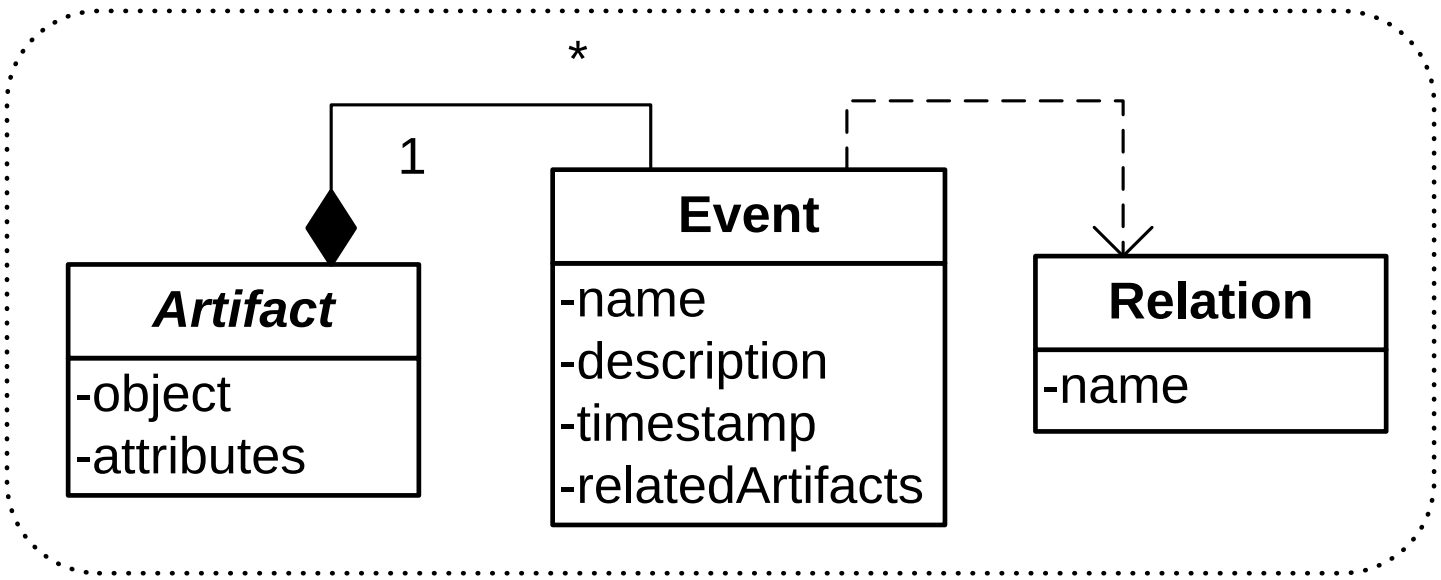
We want/need to know…

- What was done by **whom**?
- **How** did we get to what we have now?
- What **impact** do specific changes have?

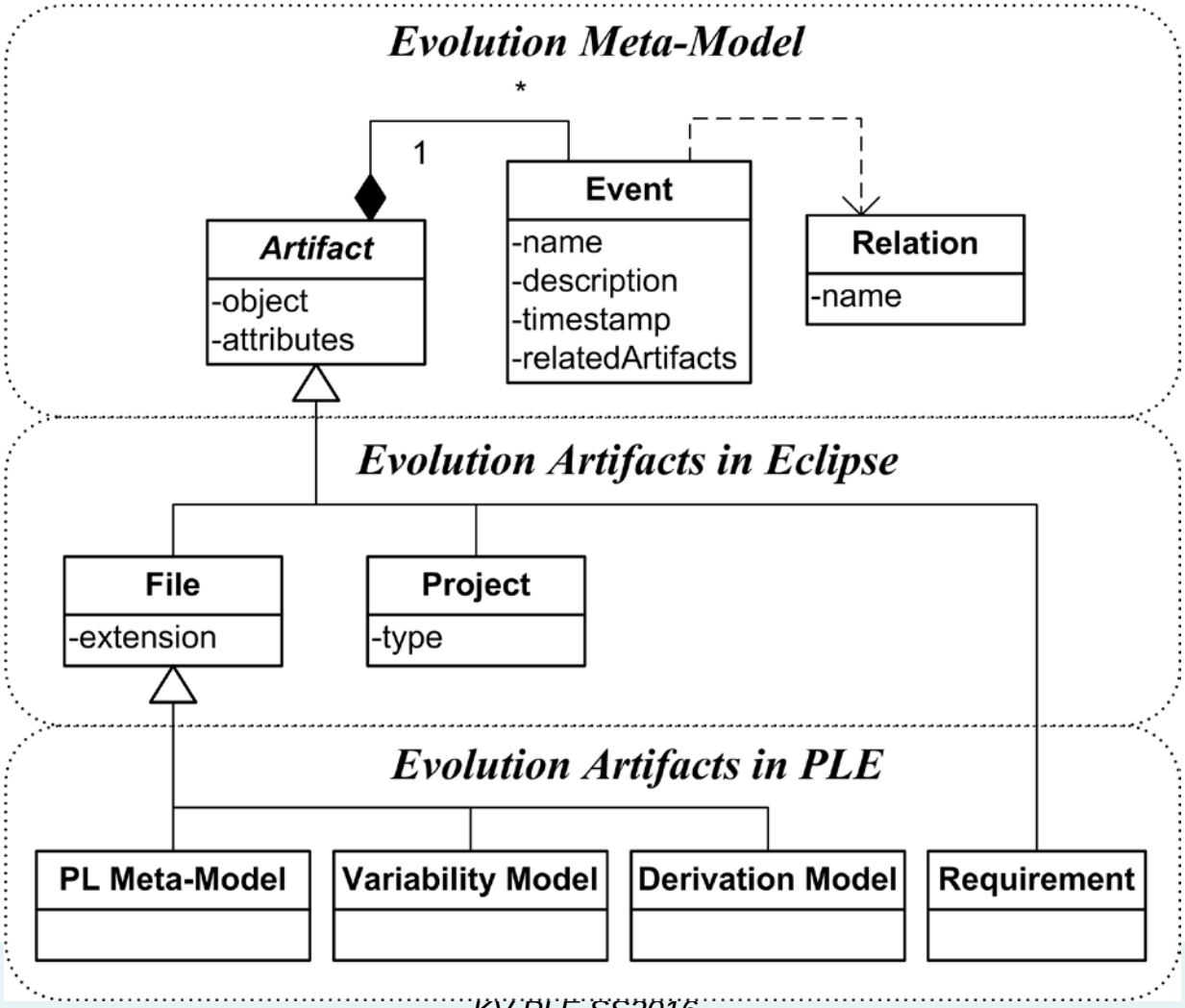
Key idea

Meta-Model for Evolution tracking of elements that:

- exist - are changing - are interrelated

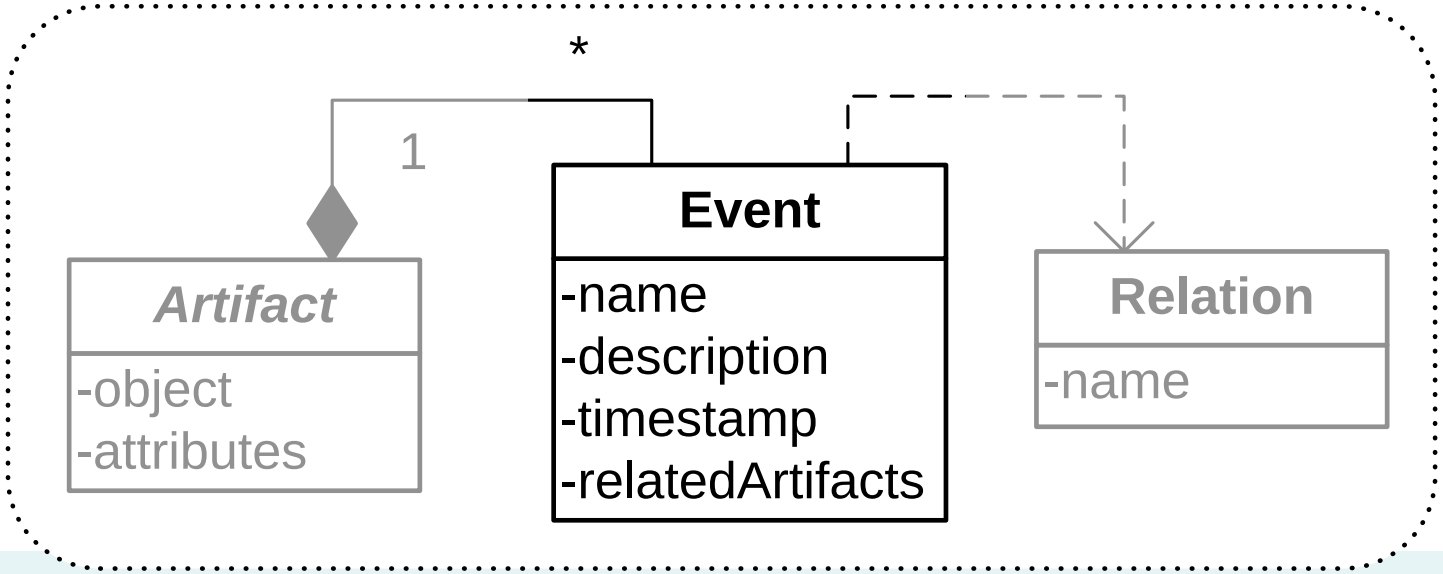


Examples of Artifacts



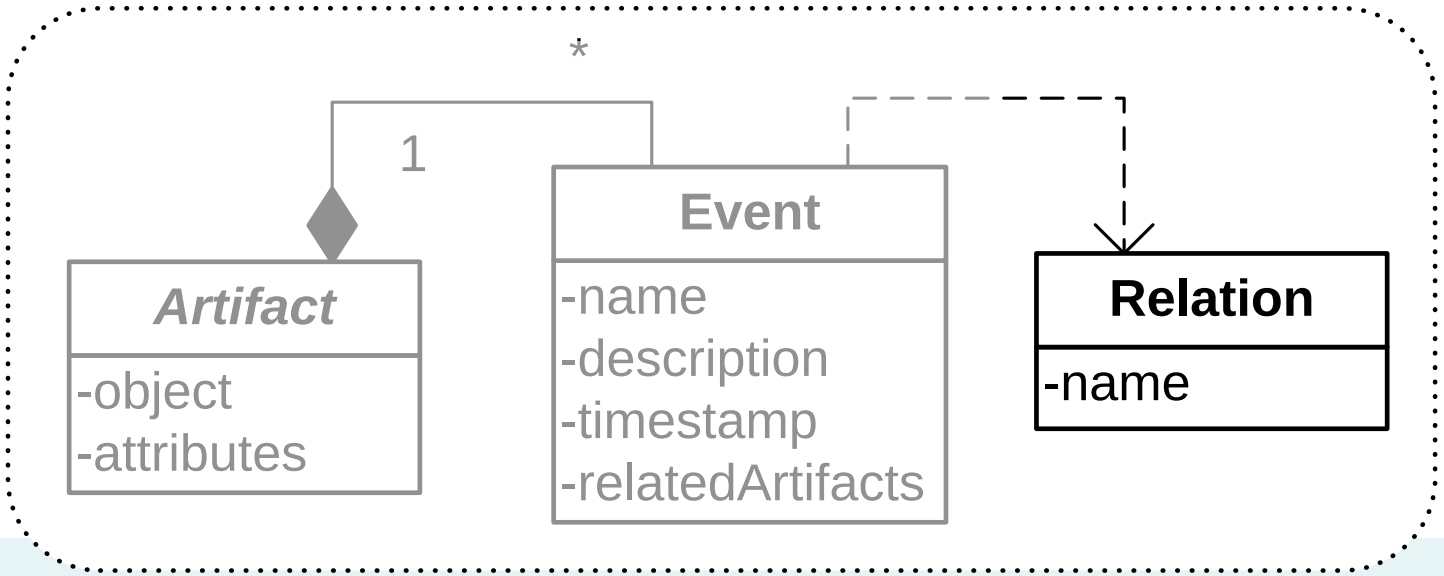
Examples of Events

- New file/project
- New variability model
- New variation point
- …

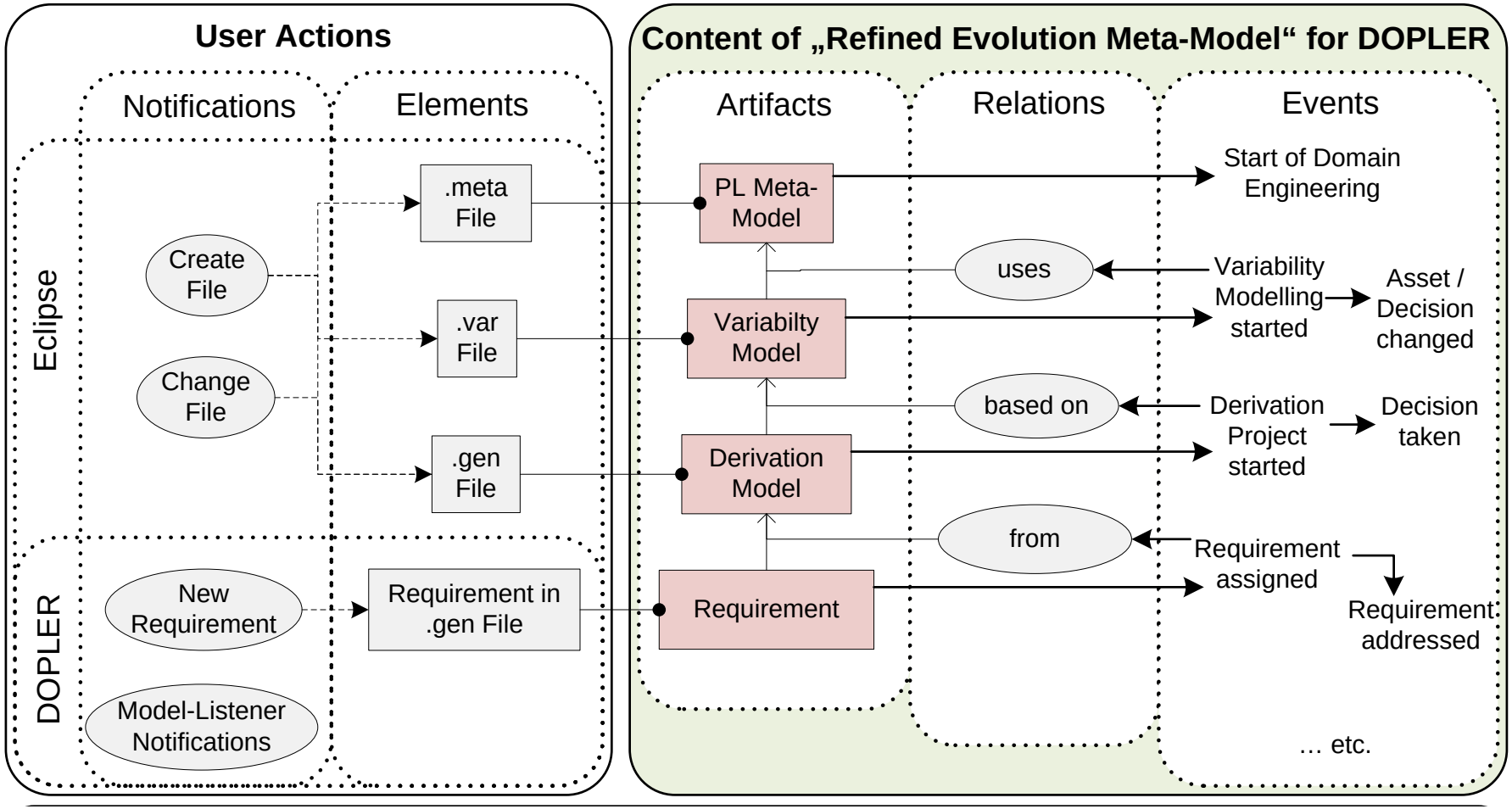


Examples of Relations

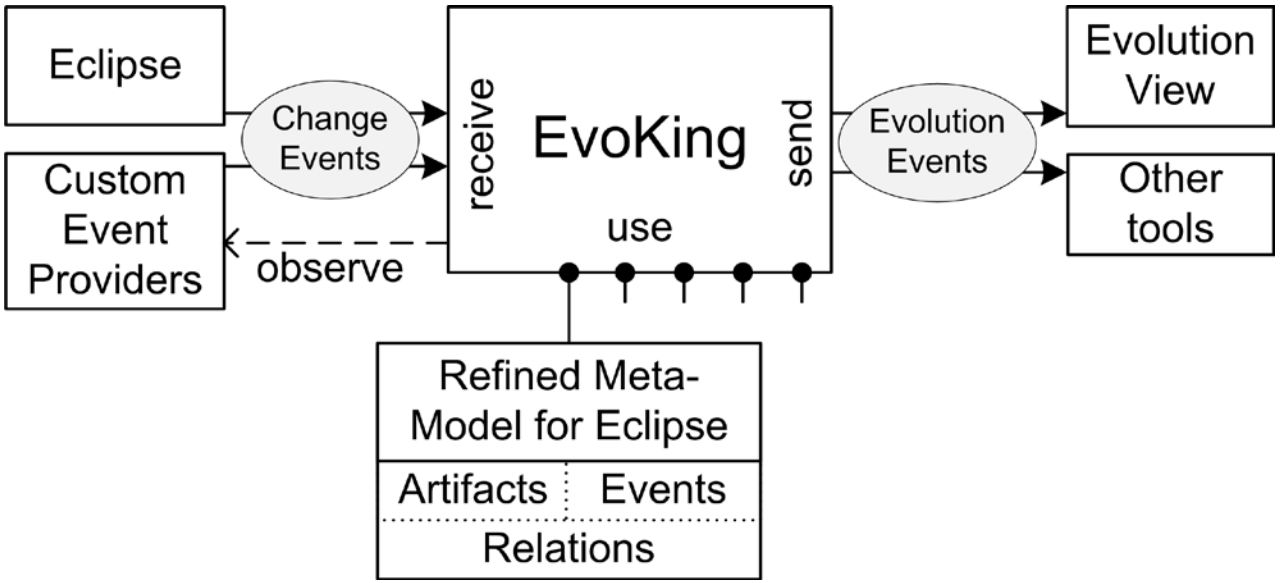
- Project to file
- File to model
- Model to model (based on, …)
- Model to model element (contains, …)
- Model element to model (change requests, …)



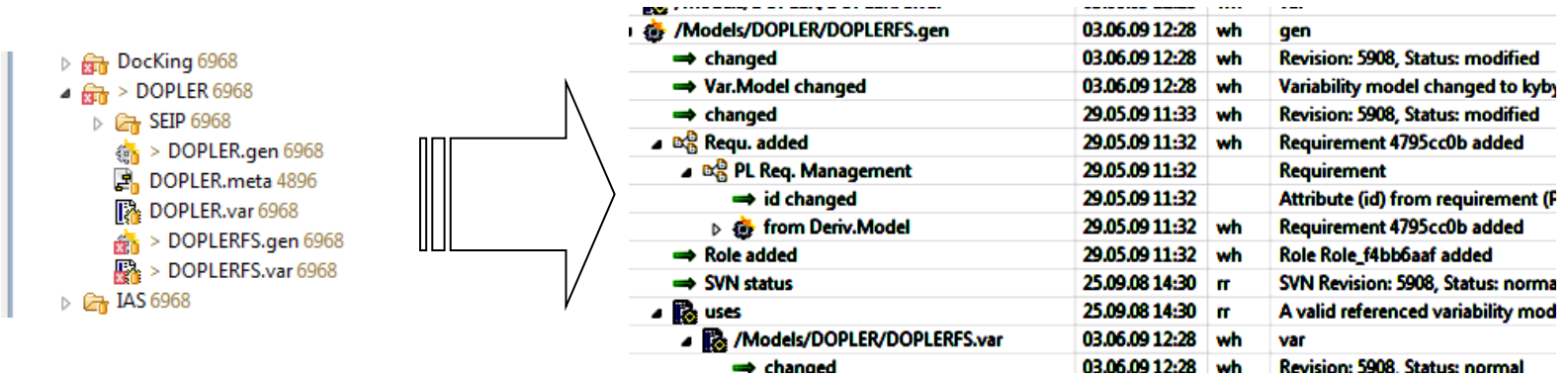
Evolution Model of DOPLER



Tool-Support: EvoKing



Conclusions



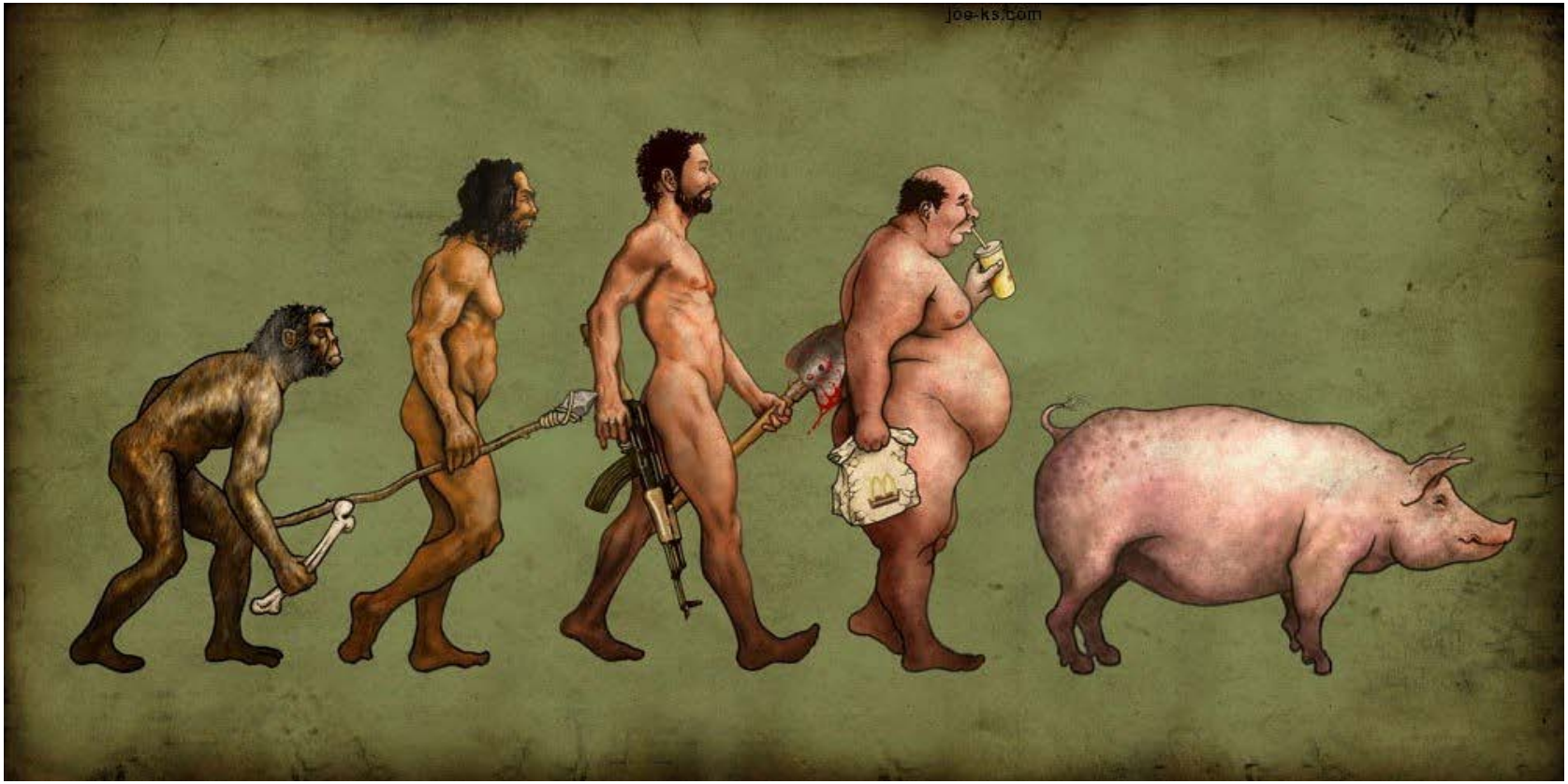
Evolution tracking with:

- Customizable multi-level change tracking
- Development history and dependency overview

Can be extended and used by:

- Implementing an evolution observer interface

Because change happens!



Next Week (1.6.)

- PL Case Studies
- Exercise 3