Component Modeling with Resemblance and Redefinition

Andrew McVeigh, April 2007

Motivations

- Initial motivation was to enable the arbitrary extension of a UML graphical case tool
  - This used a plugin-architecture and exposed some plugin-points
  - Too hard to tell the plugin-points ahead of time

Result: tool could only be extended in certain ways

Intuitions: Component Structure

- OO Issues leading to approach
  - A) Lack of “recursive” modeling approach
  - B) Code and architecture getting out of synch
  - C) Circular dependency structure of large systems
  - D) Reuse problems and backwards compatibility

- From this we get
  - A) Darwin-like / UML2 component model
  - B) A synchronization / modeling approach in the case tool
  - C) “Stratum” concept to manage “coarse” dependencies
  - D) “Resemblance” and “redefinition” to foster reuse

Reuse Intuitions

- We can express changes to a component as a series of △'s, aiding reuse
  - i.e. remove this connector, add a new one, remove a port etc.

- If a system is recursively structured as components, we can use △'s to arbitrarily change it

  Resemblance (define new component as △'s from another)

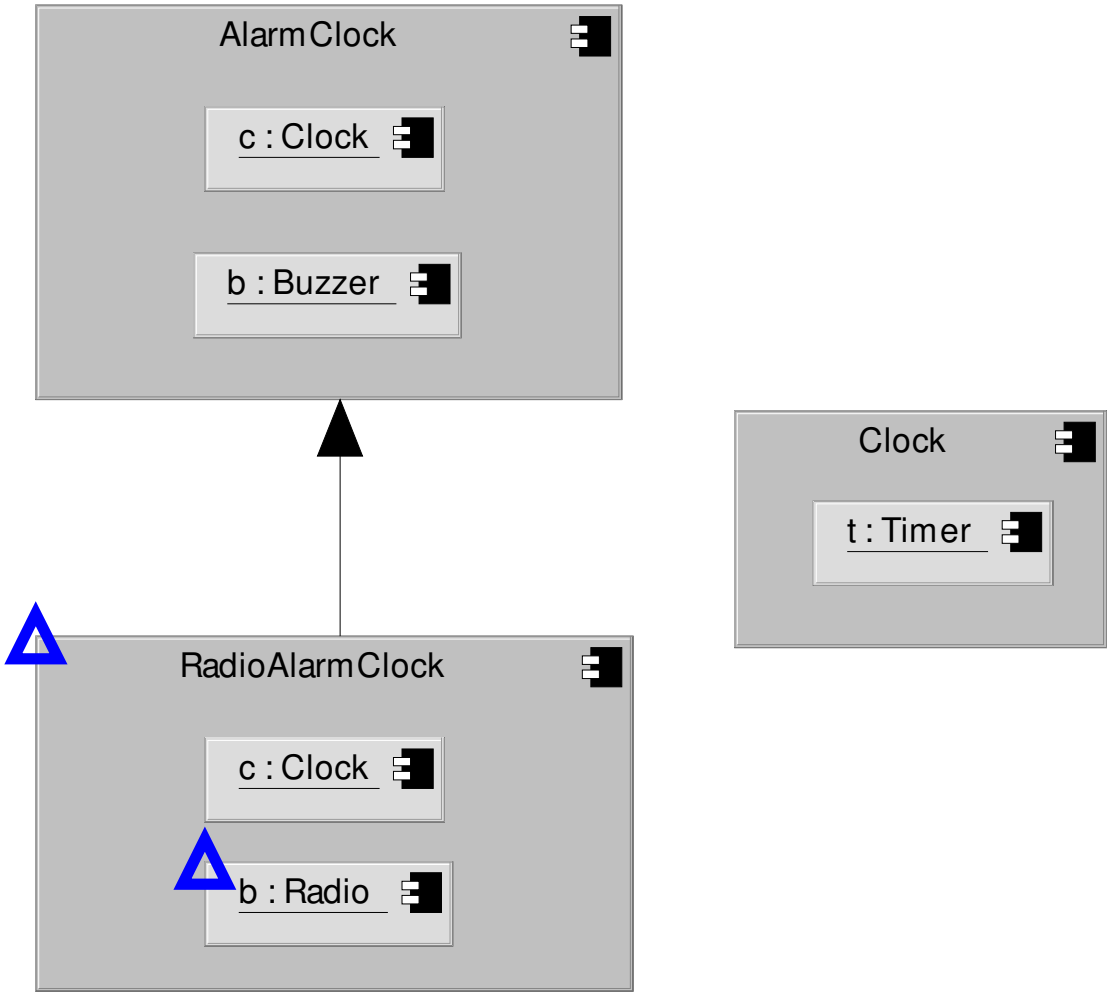
  Redefinition (change component in-place using △'s)

  - It turns out that resemblance is the fundamental construct – redefinition is resemblance + a name change

Motivating Example

A Simple Alarm Clock

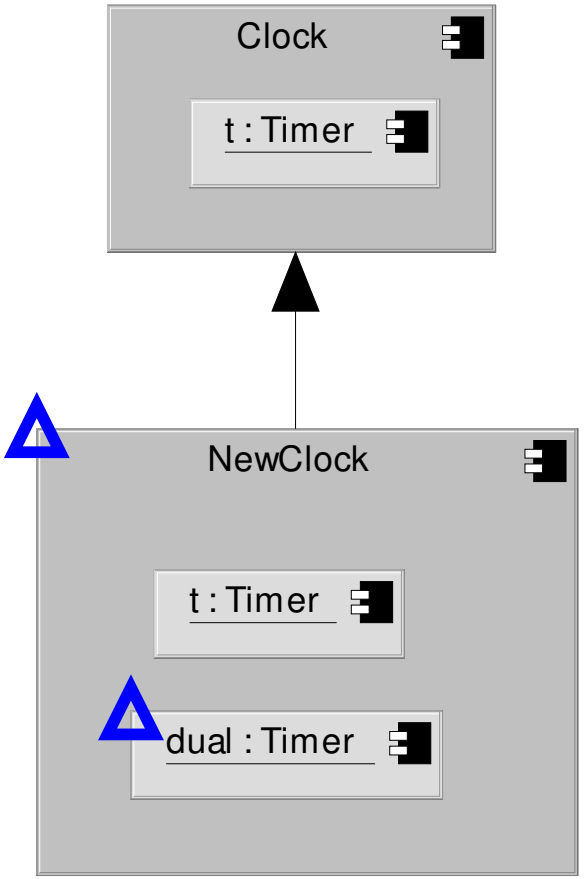
```
component AlarmClock
{
        parts:
                Clock c;
                Buzzer b;
        ...
}


component RadioAlarmClock
        resembles AlarmClock
{
        replace-parts:
                Radio b;
}
```



Changing Clock Using Resemblance (1)

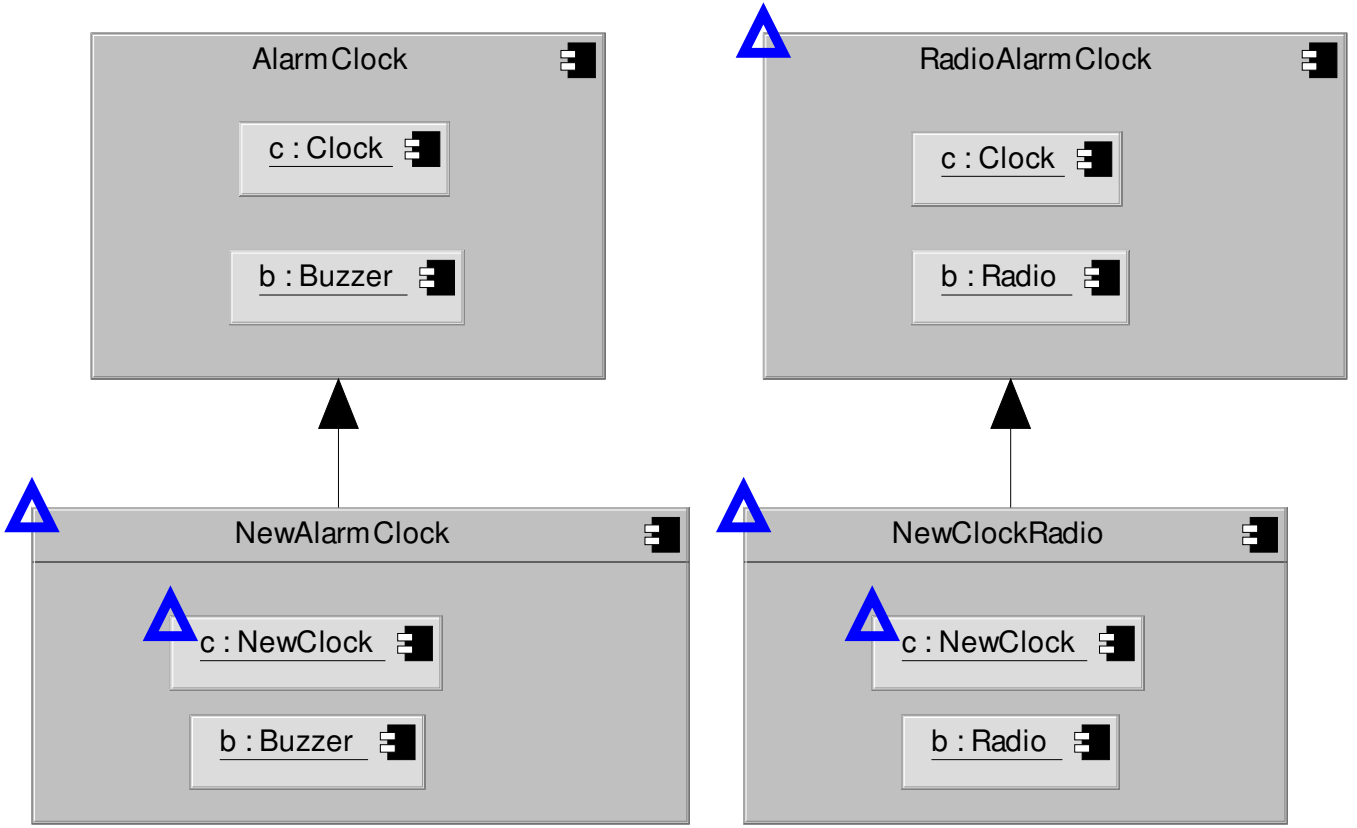
- Now, suppose we are reusing the components
  - We want to change Clock to include a new timer
  - But we can't alter the original system
  - Using resemblance we get...

```
component NewClock
        resembles Clock
{
        add-parts:
                Timer dual;
        ...
}
```



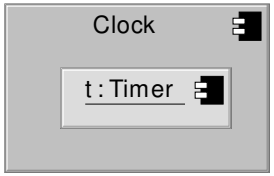
Changing Clock Using Resemblance (2)

- But then we will also need to “redefine” AlarmClock and RadioAlarmClock!
  - Updating a frequently used component results in the need for many “redefinitions”

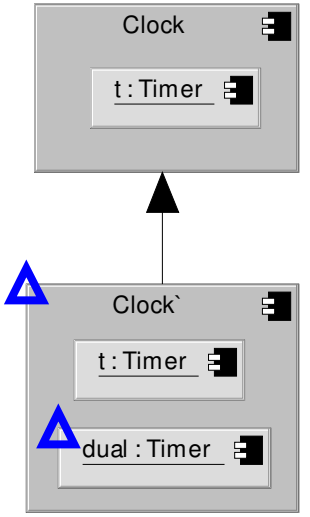


Changing Clock Using Redefinition

- Redefinition takes the name of an existing component
  - Prevents the need for lots of resemblance when adjusting a heavily used component
  - Models “evolution” of the original component




```
redefine Clock
{
        add-parts:
                Timer dual;
        ...
}
```

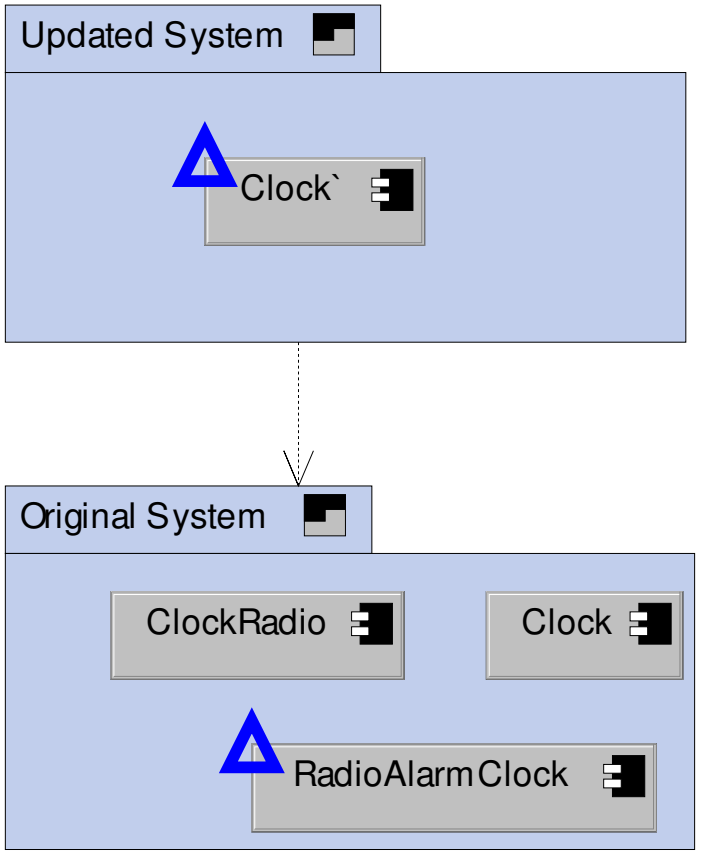


This is now used instead

Using Stratum to Manage Dependencies

What is a Stratum?

- A stratum is like a “folder” which can contain component definitions
  - The (acyclic) dependencies constrain the relationships between the components
  - For simplicity, dependencies are non-hierarchical

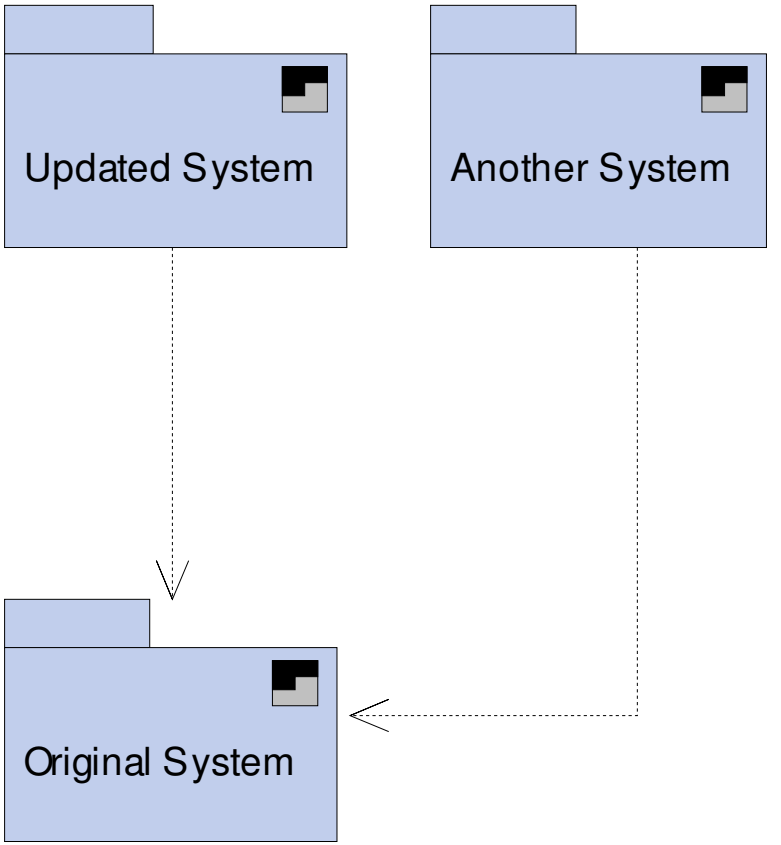


Motivation for Stratum

- The allow us to constrain dependencies at a coarse-grained level
- Allow us to group components for coarse-grained reuse
  - we can export and import independent strata
- Also act as the foundation for baselining

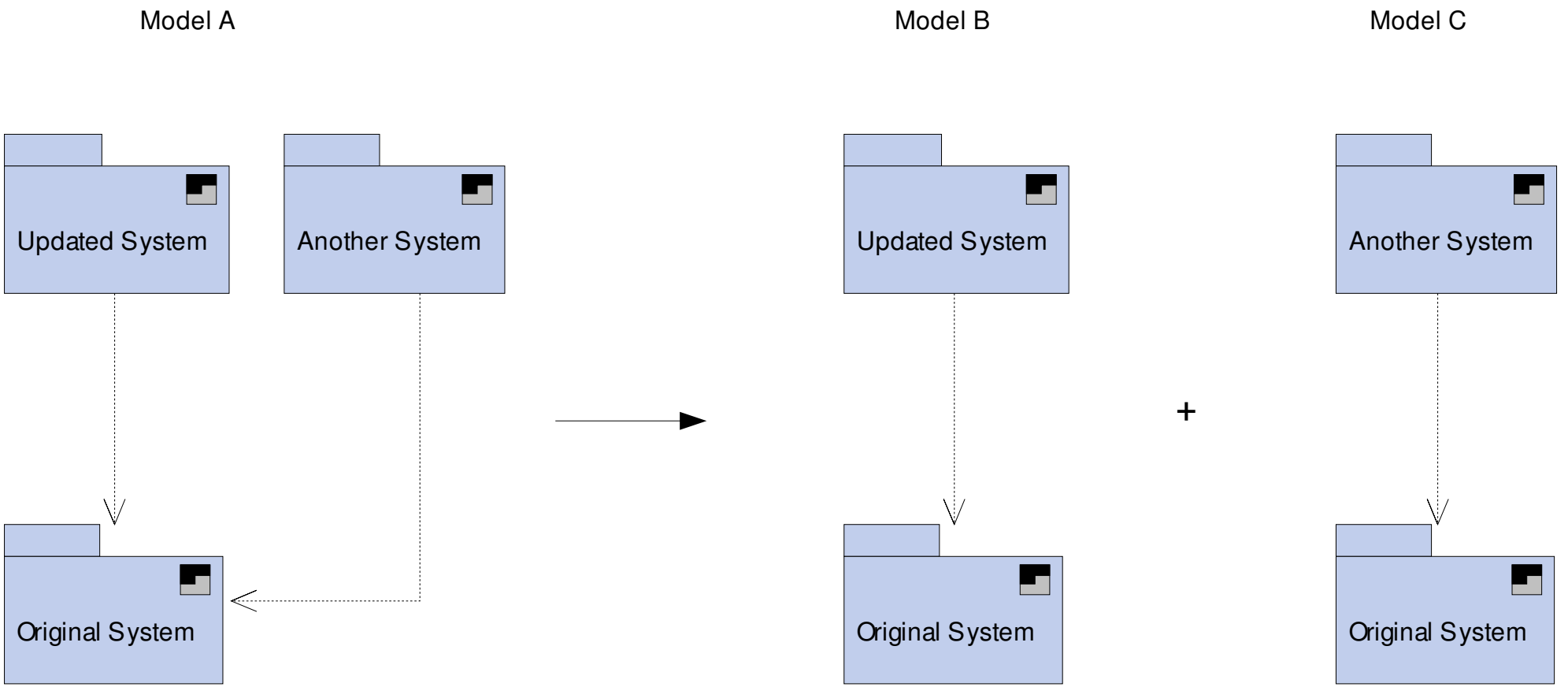
Independent Strata

- Stratum A is independent of Stratum B if A does not transitively depend on B
  - AnotherSystem is independent of UpdatedSystem
  - OriginalSystem is independent of all

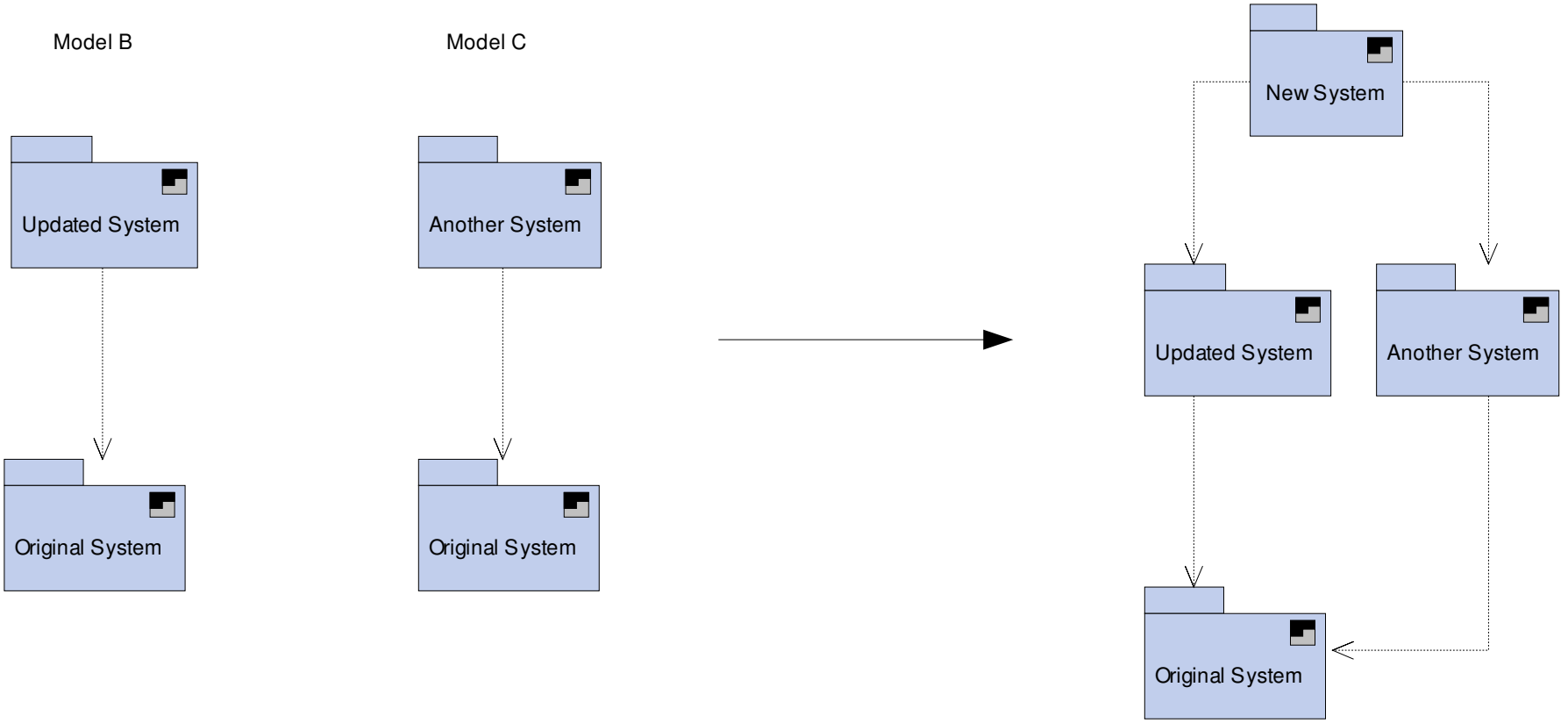


Reusing Stratum Between Models

- Strata can be exported into another model which contains the strata they depends upon

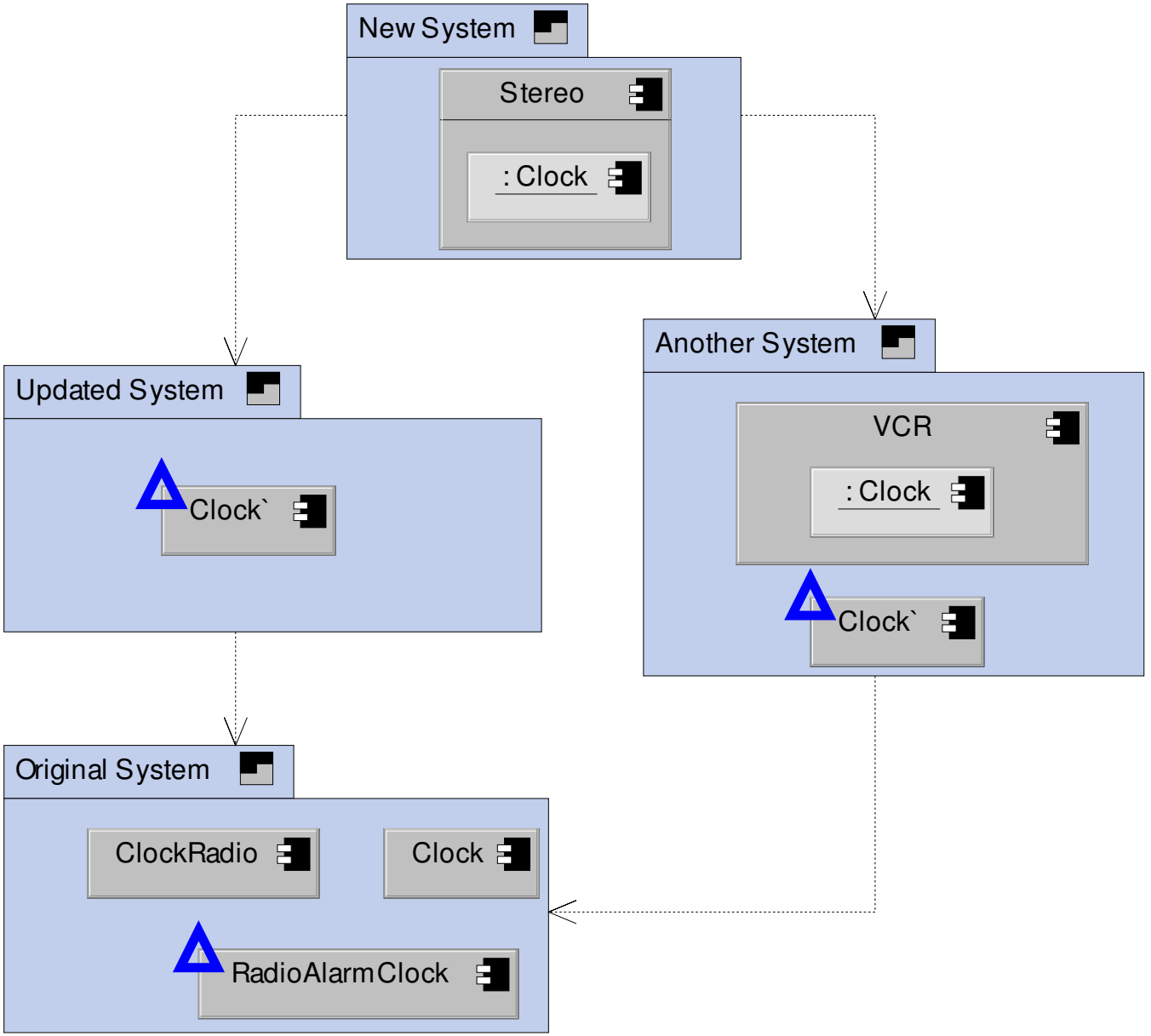


Or they can be combined...



- We do this to reuse the independently developed stratum

But combining independent stratum may lead to merge conflicts...



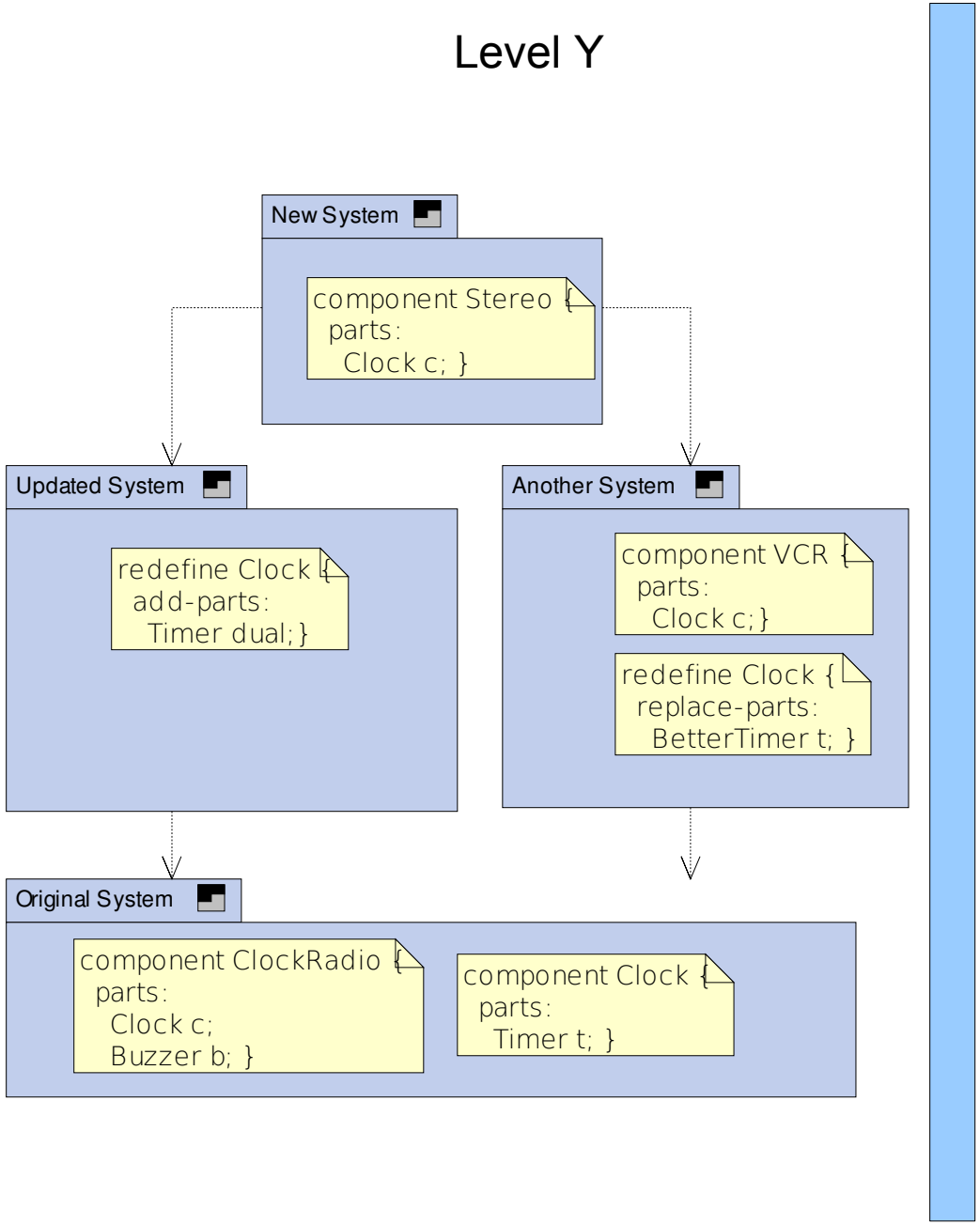
Which Clock do we use?

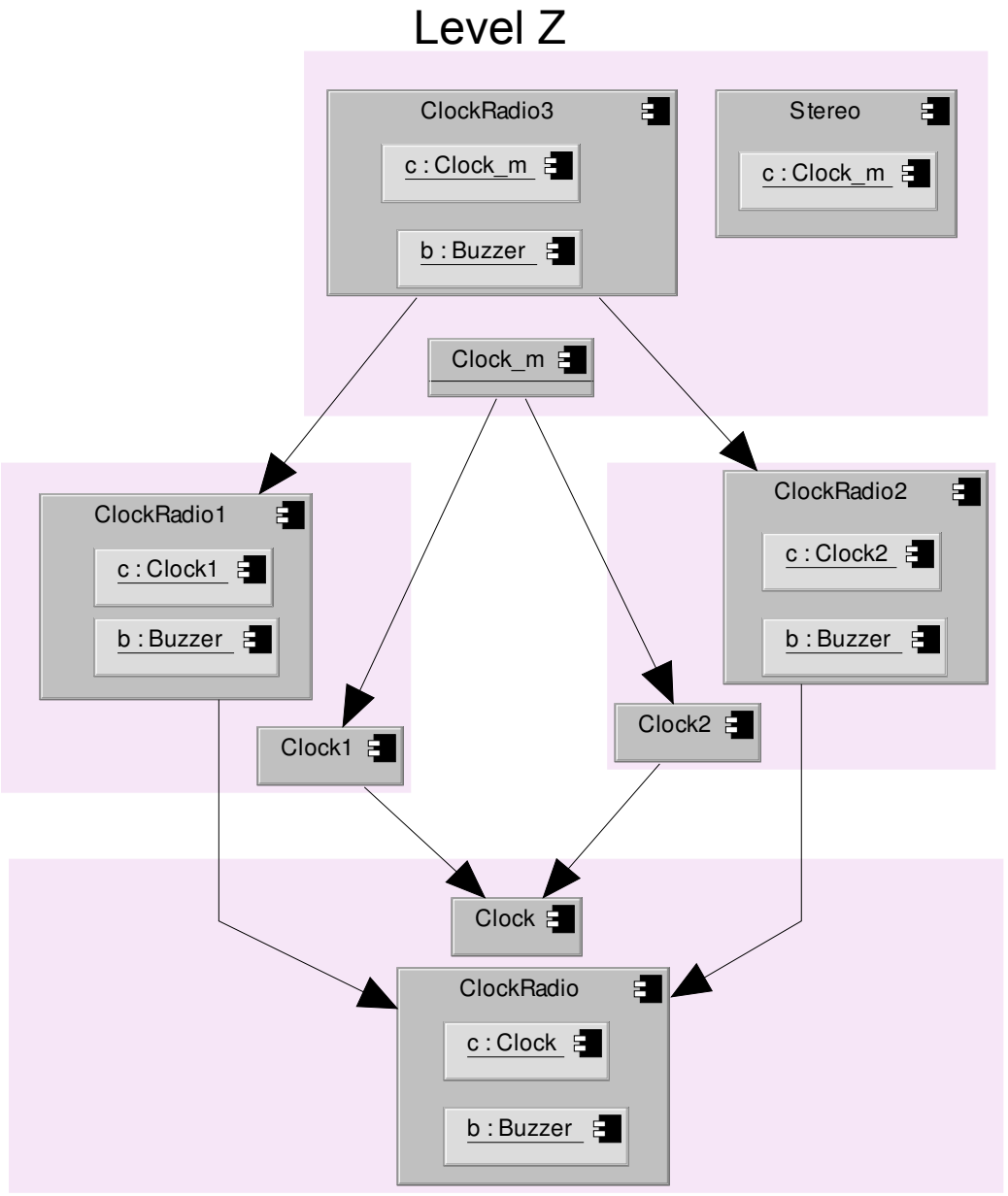
Reasoning about combined strata

OR

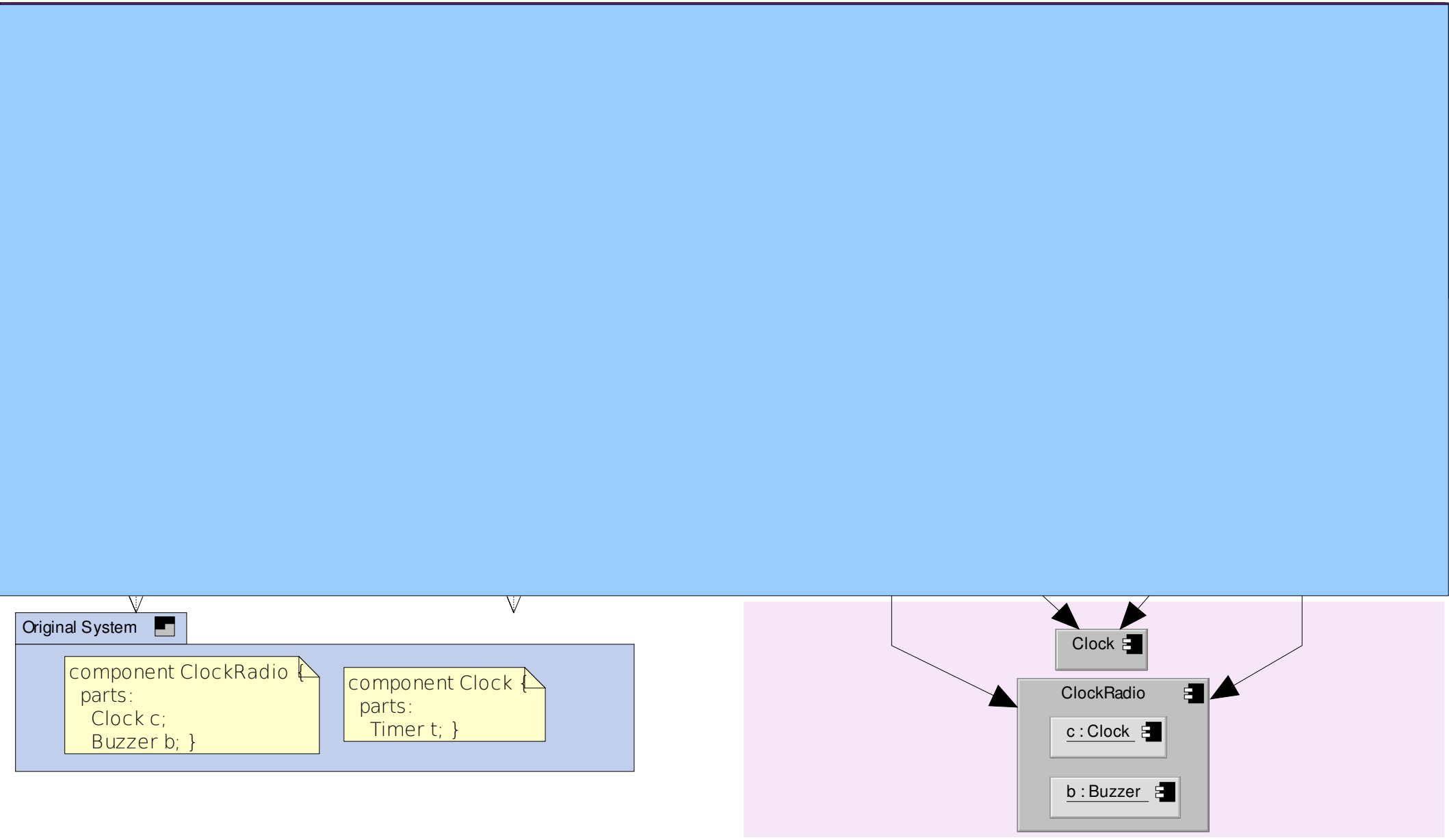
how to rewrite everything using resemblance

Transform from deltas into...



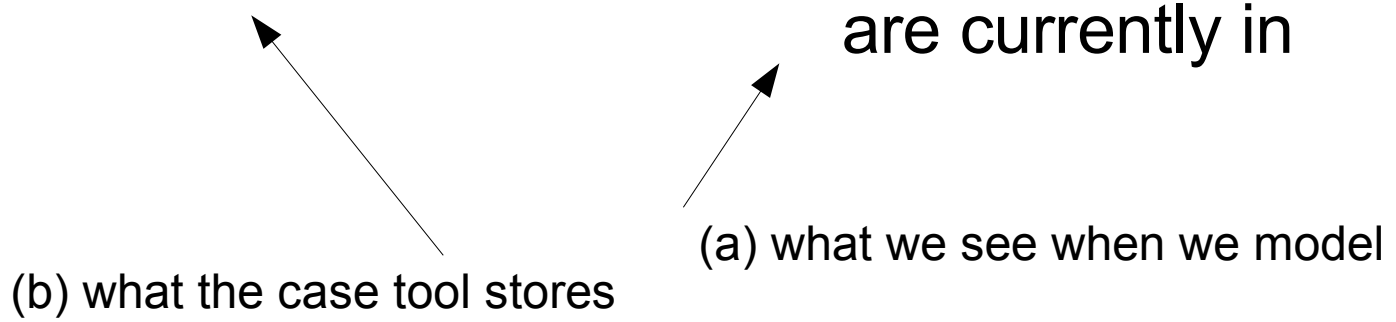


How do we interpret it?

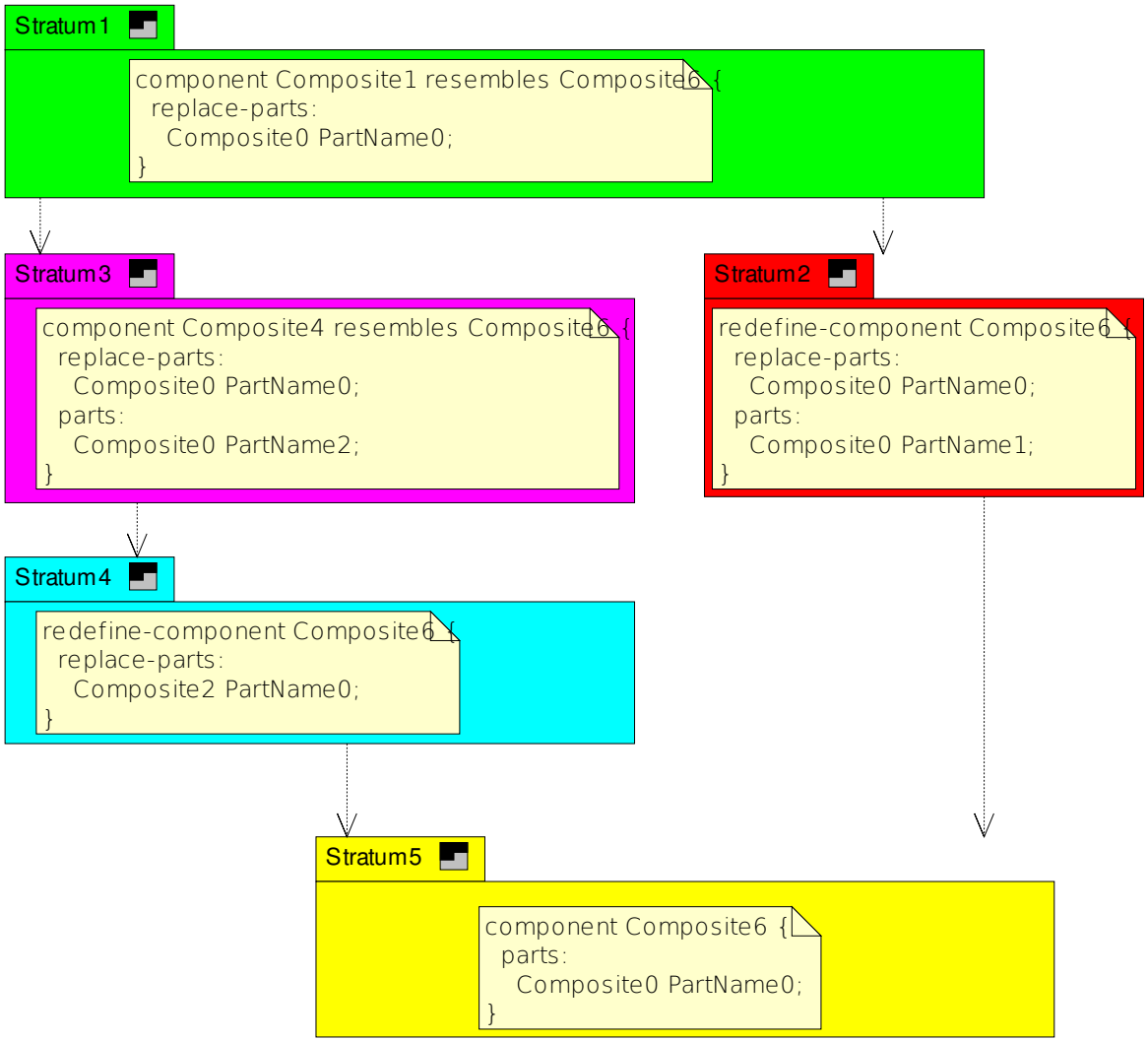


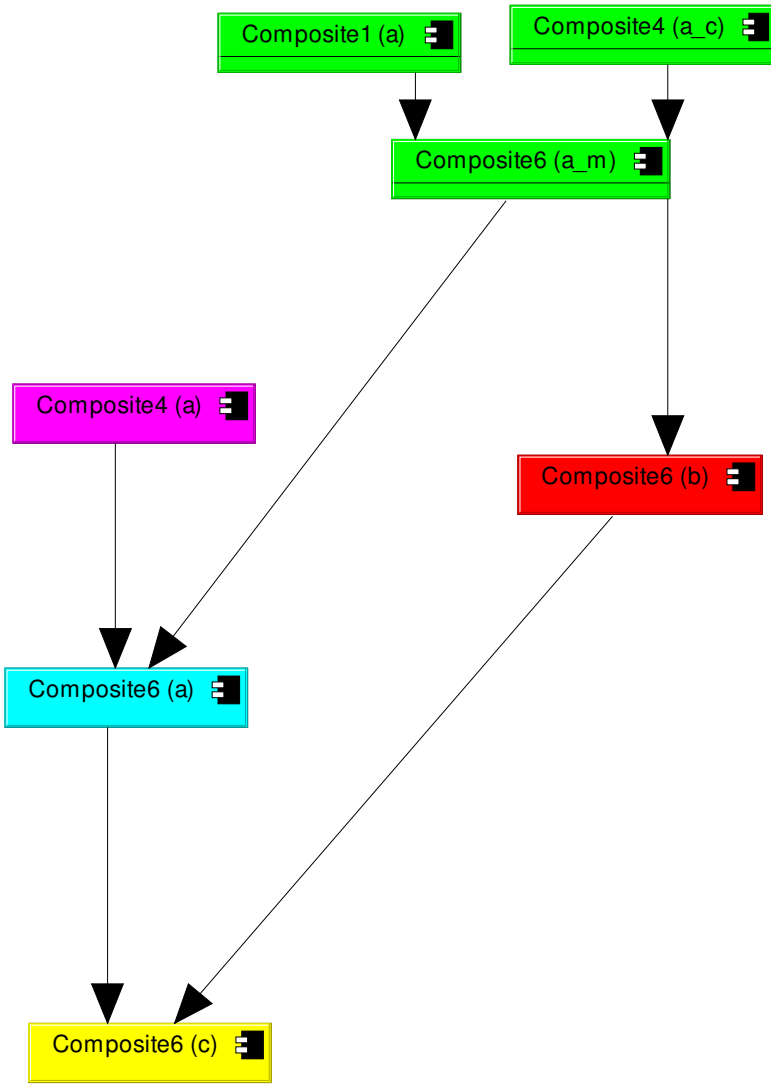
What does each level represent?

- Level Y
  - holds △'s
  - natural format of case tool database
  - some well-formedness rules here
  - stratum dependencies
- Level Z
  - used to show full graphical description
  - generated on fly as needed
  - full well formedness rules on components here
  - view shown depends on which stratum we are currently in



A Generated Model showing Merge





Current Status

Current status

- Alloy model for Y -> Z conversion done
  - Can show how merge conflicts occur
  - Shows how “stratum perspective works”
  - Can graph Y and Z in case tool
  - State space explosion – need instance size of 12 to show that all merges can be rectified (>24hrs)
- Conceptually the rest is straight forward
  - Wrapping, Replacing, Interface redefinition
  - I can show baselining informally
- But what approach?
  - Graph transforms? Smaller independent Alloy models? Theorem provers for logical model?