Detecting and resolving conflicts and interference between independently developed extensions to an architecture

April 19, 2006

1 Terminology

Model definitions (must be compatible with Eclipse definitions)

- An extension extends an architecture, by adding, adjusting or removing features. In doing so, it may evolve parts of the architecture to make it possible to add the features.
- An extension point is an explicit place in the architecture which accomodates planned extension through a required interface.
- An extension consists of a coherent set of component definitions, evolutions and merges. A component evolution adjusts a component in the architecture in some way to better accommodate a new feature. It may introduce new extension points to the architecture.
- A plugin is a degenerate form of an extension, which contains only definitions and connections to integrate new components into existing extension points.
- An extension can further expose its own extension points which can be used by other extensions which explicitly depend upon this extension.
- An extension may define or evolve code artifacts such as classes and interfaces. An extension may evolve just some of these artifacts rather than "owning" the entire code base.
- Extension A may depend on the presence of extension B. From the perspective of extension A, the architecture consists of the original architecture and the changes and new features provided by extension B.
- The constructs can model plugin architectures, frameworks and component-based architectures. The technique naturally produces extensible systems, where unplanned extension can be accommodated. In the case of unplanned extension, the level of alteration of the existing architecture determines how feasible it is to extend the system to add the features. In general, the granularity of description of the architecture determines how much alteration is required in order to add a new feature.

- The constructs are design-time rather than run-time. Reconfiguration occurs to the definition of the architecture.
- A progression is a type of extension that evolves the system forward. Rather than adding extra features, it may significantly alter the architecture. An extension is analogous to a version control branch. A progression is more like the trunk. There is no real distinction between a progression and an extension, apart from a tendency for a progression to "own" the code base and evolve significantly more code artifacts than an extension.

Feature types (can only have one)

- Deeply cross-cutting (best dealt with at a meta-level).
- Conventional.

Relative strata concepts (can only have one per stratum)
Generally, extensions are done to a base architecture. The base architecture is defined in one or more stratum and represents the starting point before the system is extended. A "progression" is another name for an extension, but where the intention is to evolve the system going forward.

- Base architecture (The main architecture being extended or progressed. The intention is to define the system.)
- Extension (An extension to a base architecture, analogous to a version control branch of the system. The intention is to add features.)
- Progression (A type of extension, analogous to the version control trunk of the system. The intention is to evolve the system.)

These concepts are largely descriptive, representing an intention rather than a fixed set of rules. An analogy is with a version control system, where a branch is often created to add a feature to an existing system or to fix a defect. The trunk represents the ongoing evolution of the system. However, sometimes the system is progressed along multiple branches, blurring the notion of a main trunk.

The terms above are also relative. An extension combined with a base architecture may together represent the base architecture for another extension. A progression combined with the base architecture may represent the new base architecture for future extensions.

Conflict and interference

- Conflict is when two extensions conflict structurally
- Interference is where two extensions conflict at a behavioural or goal level to prevent one or more extensions from accomplishing their objectives

Incompatibilities between extensions (can have one or more)

- Low-level (code related, subject to limitations that a system must have a unified view of a leaf component name, or an interface name)

- Structural (perhaps structural change is required for an extension because not all information is present at the level of change).
- Behavioural (detectable by regular protocols with non-deterministic behaviour. at the level of a component)
- Goal-level (detectable by assertions reflecting the goals of the extension? c.f. the weak invariants of the Atlee paper in telecomms. at the level of an extension)

Interaction possibilities between extensions (can only have one)

- Independent (no interaction is needed or is possible)
- Desirable (it would be desirable to enhance one or both features in the other's presence)
- Required (having the feature means that it is required to update the other features, even though it may work in isolation)
- Forbidden (both features can not be allowed to exist in the same architecture)

An extension adds, removes or modifies features of an existing architecture. Sometimes the architecture must be restructured in order to accommodate the feature. Evolution constructs exists to allow this type of alteration. This works well for an isolated extension, where the assumption is that the team responsible for the extension understands all features contained within and has made them work architecturally together.

Usually, it is necessary to combine multiple extensions in order to create a product. Two (or more) of these extensions may contain evolutions which interfere with each because they alter the architecture in incompatible ways. For instance, an extension may require the architecture to be altered in a way which is structurally incompatible with the architecture of another extension.

These constructs can model the natural process of extending and architecture where the architecture has been designed for planned extension. They also allow a system which has not been designed this way to be extended. In this way, the constructs allow for the creation of naturally extensible systems. The constructs can also model the evolution of a system.

2 Modelling extensions: definition, evolution and merge constructs

Backbone provides constructs to support for the following:

1. **Definition** - covers the creation of new components and interfaces
   (a) **Defining** a stratum (a strata depends on other strata, but does not contain them. All other constructs must be contained in a strata)

        i. A stratum can hold the original architecture, or an extension.

   (b) **Defining** a new interface

   (c) **Defining** a new leaf or composite component (possibly in terms of differences from an existing component using the is-like construct)

2. **Evolution** - covers evolution in code-related artifacts and changes to composite components

   (a) **Upversioning** an interface (incrementing the version of an interface; to reflect changes at a code level to a named interface)

        i. The new interface can be marked as a subtype of the old interface. This implies that ports that require this interface do not need to be upgraded.

        ii. This implies that the interface has been changed at a code level. Affected code-level components will require change. There can only be a single code-level version at a time and adapters are not possible.

        iii. Carries limitations for combining extensions.

   (b) **Upversioning** a leaf component. (incrementing the version of an existing leaf component to reflect changes at a code level)

        i. Is-like may be used to indicate delta changes to an existing leaf component.

        ii. This implies that the component has been changed at a code level and combined with the existing system. There can only be a single code-level version at a time.

        iii. Carries limitations for combining extensions.

   (c) **Supplanting** an interface (globally replacing a code-level interface with another code-level interface and taking its identity)

        i. Can define a subtype relation between the old and new interfaces.

        ii. Code-level requires of old interfaces do not need to be altered if there is a subtype relation.

        iii. Can define a global adapter for converting old requires and provides. Adapters are not required for old requires where there is a subtype relation.

   (d) **Supplanting** a component (globally replacing a leaf or composite component with a new composite)

        i. If a component supplants a leaf, the new composite may wrap it and delegate to it.

        ii. If a leaf supplants another leaf, in the code it may inherit from it.

        iii. The new composite may use the is-like construct to structurally inherit, and adjust, the features of another composite component.

        iv. Supplanting and is-like work together to allow delta changes to be expressed to a component, reflecting evolutionary change, even at the leaf level.

        v. Supplanting implies taking on the identity of another component.
        vi. Provide a way to refer to the unsupplanted component, i.e. the component before the strata redefined it.

3. **Merging** - combining multiple, independently developed extensions and resolving conflicts
    (a) **Detecting and resolving** conflicts between multiple versions of a single interface.
        i. Must definitively upversion (with a code artifact) and either declare to be a subtype of all existing versions, or define a new version and replace the old versions in the architecture. Only one version can exist for a given code name, which implies code-level merging.
    (b) **Detecting and resolving** conflicts between multiple versions of an existing leaf component.
        i. Must definitively upversion (with a code artifact), or potentially supplant parts of the architecture to resolve.
        ii. Avoid this by reducing the need to monolithically replace through granular decomposition into small leaves.
    (c) **Detecting, merging** and **resolving** conflicts between multiple supplantings of a leaf or composite component.
        i. Also affects any components that compose this.
        ii. Structure and/or interfaces may be affected.
        iii. Need to define interference and conflict
    (d) **Detecting, merging and resolving** interference at a behavioural level

The constructs cover defining an architecture, extending an architecture through the addition of new features and evolution (possibly at the code level), and the merging of independently developed extensions into a coherent system. The need for merging is a consequence of allowing independent extensions of an architecture, which must be later combined to produce a single product. When combining extensions, it is possible to use the constructs to integrate the features of both to provide any desired interaction.

Composition is supported through the definition constructs. Decomposition is supported through the definition and evolution constructs which allow an existing component to be replaced by a broken down set of components which have been composed back into something compatible with the original.

3 Stereotypes as annotations

Stereotypes can be applied to components, interfaces, properties, parts, connections and ports. The stereotype concept has been expanded to allow a list of tagged values to follow the stereotype name, giving a similar power to annotations in Java. [Java ref, UML-F ref].

4 Pattern constructs

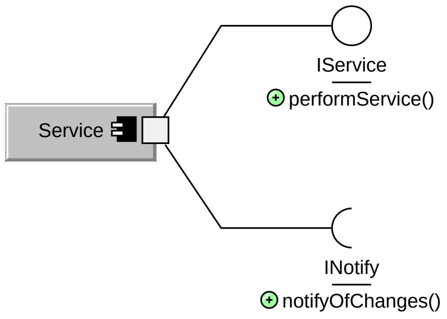
Certain patterns occur again and again when defining extensions. As a result, Backbone reifies some of these patterns in order to capture the intent and allow this to be taken into account when combining extensions.

4.1 The multiple-client adapter pattern

When extending an architecture, it is common to come across a service component which expects that it will only be used by a single client. In order to use this from multiple clients, the service component must be altered to support extra clients, whilst still preserving the illusion to existing clients that they are the sole client. In some cases, this involves adapting required interfaces (for notification) and provided interfaces (for services). The pattern encapsulates the service component and introduces a mode, representing the current client to have exclusive access.

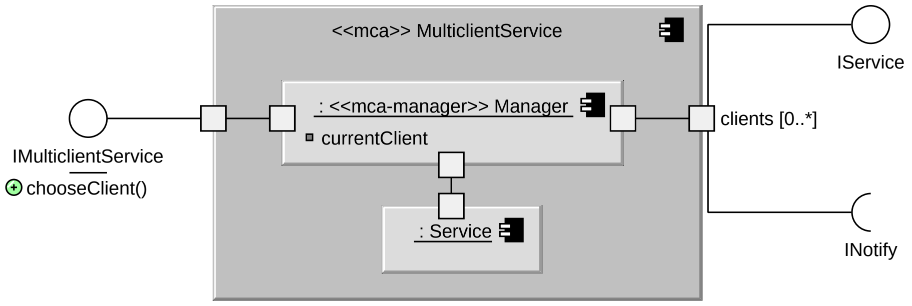
This is represented by a stereotype of «mca» on the adapting component. This enforces the interfaces that must be present, although it does not dictate the logic of the manager component. An example of a single client is shown below.

Figure 1: A single client with single service and notification interfaces



This can be adapted by the following component. The stereotype «mca» (multi-client adapter) is used for the composite. The manager must be marked with the $<<$mca-manager$>>$ stereotype.

Figure 2: Adapting a single client service for multiple clients



Sometimes it is necessary for an adapter to remember some state of the service for a client. For instance, if we are creating a multi-client LED display from a single-client one, it will be necessary to store the digits for the previous client so that when the client is re-chosen, the display can be restored correctly. This is necessary to support the illusion that existing clients “own” the service. In this case, the «mca-stateful» stereotype is used for the adapter. The name of the adapter is contrained to be “Multiclient” as a prefix in front of the original service name.

The stereotype «mc» can be applied to an upversioned service component to indicate that it supersedes the multi-client adapters. The «mc-stateful» stereotype indicates that this remembers some aspect of client state for when a client is chosen again.

5 Examples of constructs

A stratum contains definitions, evolution and merging constructs. A stratum depends on other stratum, and the constructs within may only access constructs contained in stratum that is visible to this stratum. If a stratum is relaxed, it exposes its dependent strata to any strata depending on it.

5.1 Examples of definitions

Listing 1: Defining a stratum

```
// defining a stratum
stratum X
 depends-on Y, Z
 is-relaxed;
```

An interface must have an associated code-level interface. Any subtype relations must be explicitly declared.

Listing 2: Defining an interface

```
define-interface I associated-with-code-level CI
                   is-subtype-of SI1, SI2...
{
  // reserved: possibly for behavioural specification?
}
```

A leaf component must have an associated code-level class. It can define ports and properties which must match up with the code-level class counterpart.

Listing 3: Defining a leaf component

```
define-leaf-component LC associated-with-code-level C
{
  properties:
    String name(test);
  ports:
    portA provides interfaceP requires interfaceR;
}
```

A leaf component may declare itself to be like another leaf component. The is-like construct allows one component to be defined in terms of another. Properties and ports can be added, removed and redefined.

Listing 4: Defining a leaf component to be like another leaf

```
define-leaf-component LC2 associated-with-code-level C2
                         is-like L
{
  properties:
    String title;
  redefine-properties:
    String name(new test);

  ports:
    portB provides interface P1;
  delete-ports:
    portA;
}
```

A composite component is a compositional construct with no associated code-level class. It can additionally define parts and connections, in addition to properties and ports. Each part can have its properties set. If the properties are not set, they get the default value (if any) specified in the parts component type.

Listing 5: Defining a leaf component to be like another leaf

```
define-composite-component CC
{
  properties:
    String title;

  ports:
    portA provides interfaceP requires interfaceR;

  parts:
    LC leaf
      set name(title);

  connections:
    c joins portA to portA@leaf;
}
```

Composite components may be declared like another composite, again using the is-like construct. This acts as a form of structural inheritance, and the composite can redefine any aspect of the original composite. Properties, ports, parts and connections can be added, removed or redefined (an implicit add and remove).

5.2 Examples of evolution constructs

Upversioning a leaf interface relies on a new interface definition, which reflects the new version of the interface. This new definition can then upversion the old definition. The original definition of the interface must reside in a different stratum. No upversioning can occur within the same stratum as it is assumed that as a coherent entity, the extension will have no need to upversion any construct defined within. The new interface is accessible through its name (NewI) and the name it has upversioned. The previous interface definition can be referred to by placing [old] before the name. i.e. [old]I. Each interface can only be upversioned once in a stratum.

Listing 6: Upversioning an interface

```
define-interface NewI is-associated-with-code-level CI
{
}

upversion-interface I using NewI;
```

Upversioning of a leaf component is similar. The upversion-leaf-component replaces an old version of a component with a new version. The previous version is still accessible by prefixing [old] as with interfaces. Because the definition refers to the old version of LC, it must be explicitly marked this way. To not do this in this case would result in an error, as NewLC would be effectively replacing itself. The upversion below reflects that a new code-level property has been added to component LC.

Listing 7: Upversioning a leaf component

```
define-leaf-component NewLC associated-with-code-level C
                            is-like [old]LC
{
  properties:
    String title;
}

upversion-leaf-component LC using NewLC
```

Components can also be supplanted. Again, [old] allows direct reference to the previous definitions. A maxmimum of one supplanting of each component is allowed within a stratum, and any components supplanted cannot be defined in the stratum doing the supplanting. The following listing shows a composite supplanting a leaf.

Listing 8: Upversioning a leaf component

```
define-composite-component NewLC
{
  ...
}

supplant-component L using NewLC
```

All combinations of this construct are supported: acomposite can supplant a composite, a leaf can supplant a leaf, a leaf can supplant a composite, and composite can supplant a leaf. A stratum may both upversion a leaf and supplant it, if necessary.

6 Interaction with version control

How does it work in allowing someone to stay on an old version of the system, but pick up some new changes? Idea is to interact with version control allowing advantages of both approaches.

- Idea of baselining to form a definitive architecture which rolls up evolutions into definitions. Baseline may not be most current version, but may "Roll up part of the tail" to reflect older versions which are no longer used.

7 Comparing and contrasting with existing mechanisms

- With frameworks and plugin architectures (pros: BB allows for unplanned extension, supports fine and coarse grained composition, higher level semantic checks)
- With OO - automatic enforcement and monitoring of dependencies.
- With versioning systems (c.f. Mae. Must explain also how this ties into a version control system)
- With mutiple dimensions of concerns. i.e. aspects and Batory feature combination.
- With OSGI (c.f. upgrading etc)
- With runtime reconfiguration

Pros and cons of approach.
Advantages?

- Makes more flexible plugin systems where full source code is not available. Lowers burden of comprehensive extension.
- Separation of architecture from code allows simpler extension.
- Higher semantic level can involve protocol and property checks.
- Expressing change as deltas allows a fine grained form of change which doesn't imply source ownership of entire component. Can still pick up changes to base.
- As architecture becomes finer grained, cost of change is lower as only small leaf components require change.

Limitations?

- Cross-cutting not handled well?
- Code related to do with upversioning components and interfaces - must have a system-wide view of a code interface or leaf. How about defining new interfaces as the new version allowing multiple to coexist? Defining a new class?

8 Implementation

- Can this be implemented on top of an OSGI base? Possible later runtime reconfiguration? Is runtime reconfiguration too much, and we don't want to complicate the picture by introducing it? What about runtime application of extensions? Future work? :-)
- What about implementation in a dynamic language like Python? Does this ease version problems? Interfaces specified using BB?

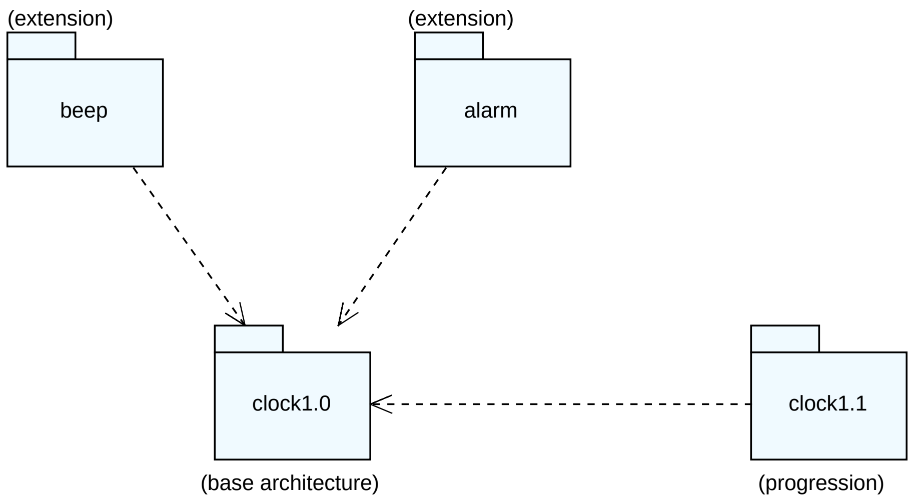
9 Clock example

The following example presents architecture of a digital clock. This provides a basic digital readout of the time and updates every minute.

Two extensions are presented that extend the architecture by adding a beep on each hour, and an alarm. These extensions are developed independently. A progression is then defined, evolving the original clock architecture to include a feature where the time is automatically adjusted by listening to a radio signal.

The figure below shows each of the strata containing the base architecture, the two extensions and the progression. Each extension will work when combined with the clock1.0 base architecture. Note that the bracketed terms are relative concepts - for another extension, the clock1.0 and alarm strata could represent the base architecture.

Figure 3: The core clock architecture, two extensions and a progression.



The table below shows that seven possible strata combinations are possible:

Table 1: Mapping system features onto combinations of strata

System features	**clock1.0**	**clock1.1**	**beep**	**alarm**
clock	*			
clock + beep	*		*	
clock + beep + alarm	*		*	*
updating clock	*	*		
updating clock + beep	*	*	*	
updating clock + alarm	*	*		*
updating clock + beep + alarm	*	*	*	*

In an ideal world, the extensions would combine to produce a coherent system where all the combined features behave as intended. As the example will show, in general and in this situation this is not the case due to a number of issues. The issues that prevent the correct combination of extensions are:

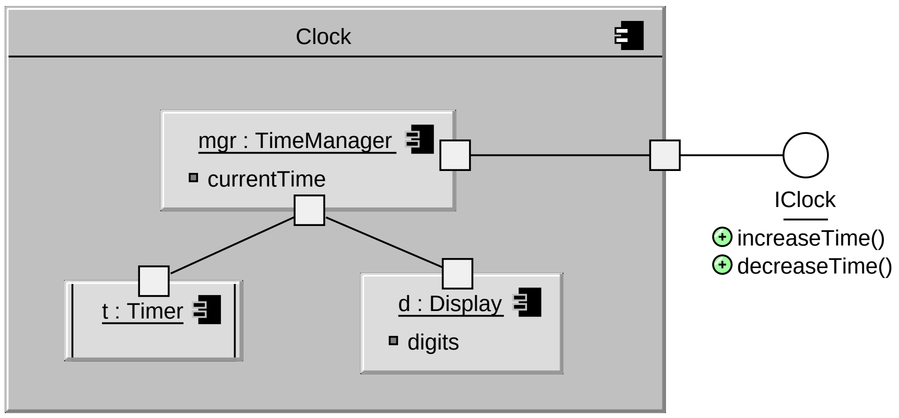
1. Structural issues
   This situation occurs when one extension makes structural changes to a component in the base architecture that overlap and are incompatible with the structural changes to that component by another extension. In the example below, both alarm and hourly-beep adjust the Timer component in incompatible ways to add required functionality. Combining extensions implies we must arrive at a consolidated structural view that works for both extensions. Structural issues also arise where each extension has added a required piece of infrastructure. When the extensions are combined, we have two or more equivalent parts. In most cases it is desirable or even necessary to combine the multiple copies of infrastructure into a single, unified entity.

2. Protocol issues
   This situation occurs when two extensions use a shared component in a way which violates its protocol, even though individually the extensions respect the protocol. An example of this is when a beep is issued whilst the alarm is sounding, even though the protocol of the tone generator stipulates that this is not allowed to happen. This situation must be remedied by mediating between the multiple users of the shared resource.

3. Conflicting Goals
   This occurs when the goal of an extension is frustrated or rendered unreachable in the presence of another extension. The example that will be explored below is when an hourly beep terminates the alarm tone early, violating the alarm's intention to sound for a given time interval. Although all protocols have been respected, the higher level behaviour of the extensions has been frustrated by the interaction of the two extensions.

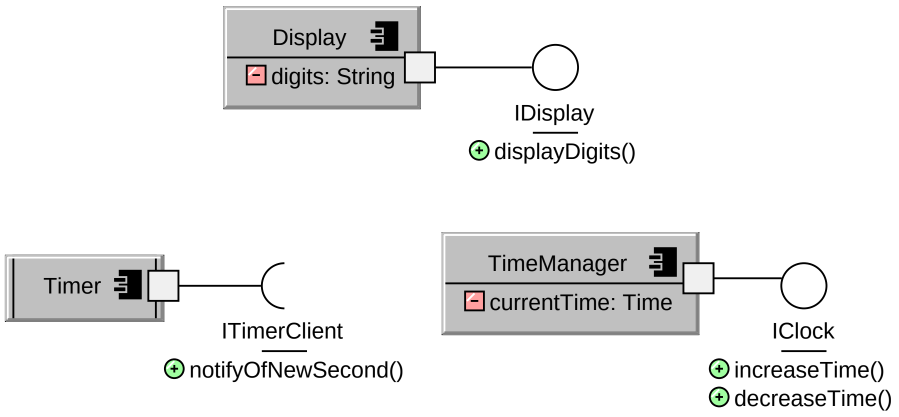
9.1 The base clock architecture

Figure 4: The basic clock as core architecture (clock1.0)



The base architecture describes a basic digital clock. It has a timer, which notifies a single client each second and a display, which can show a series of 8 digits for hh:mm:ss time format. The TimeManager holds the current time and acts as a client to the timer. Each second it increases the time and displays it. The internal components are shown in the following diagram. They are all leaf components, as evidenced by the black prongs on the component icons. Composites have white prongs, as with Clock above. Timer is active, and has its own thread of control for notifying the client. Note that only one client is supported.

Figure 5: The internal components of the clock

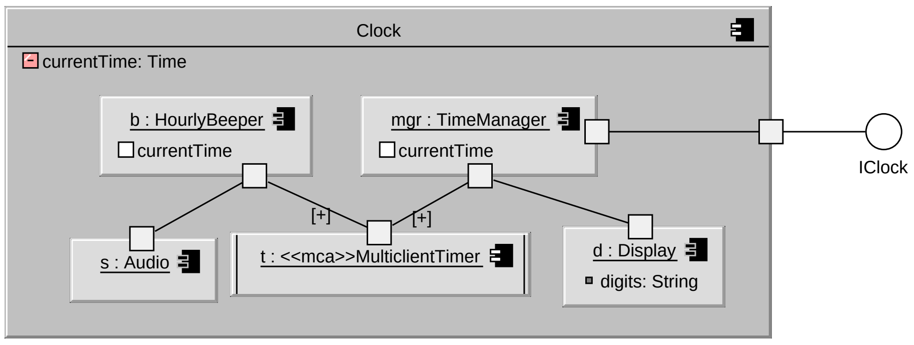


The IClock interface allows the time to be adjusted. The buttons that allow this are not part of this architecture.

9.2 Adding an hourly beep

The beep stratum is an extension to allow the clock to beep every hour. The figure below shows the evolved Clock component that achieves this

Figure 6: The evolved Clock component to add an hourly beep (beep)

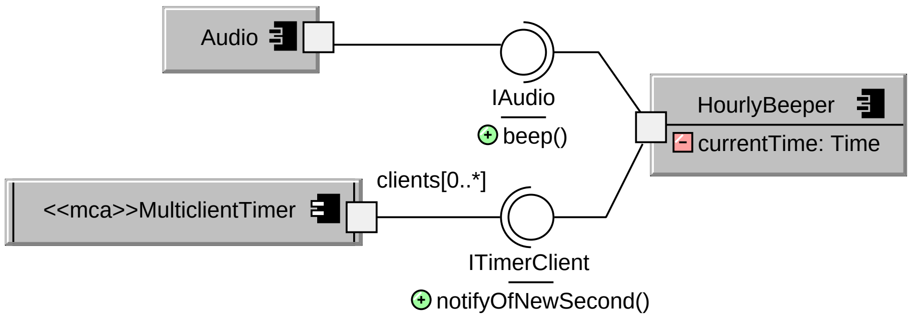


The HourlyBeeper needs to be notified of timer updates, but the problem is that the existing Timer only supports a single client. As a result the beep extension introduces a component called MulticlientTimer. As it turns out, it is a prevalent requirement to

Audio is a component that produces a beep of a fixed pitch, at a fixed volume, for a duration of 1 second. HourlyBeeper is a control class that monitors the current time (based on being a client of MultiTimer) and sounds the been when the hour has been reached.

The internal components are shown below.

Figure 7: The internal components of the beeping clock



The two main problems facing the designer of this extension were:

1. The timer could only have one client
2. The current time information was not available via the TimeManager

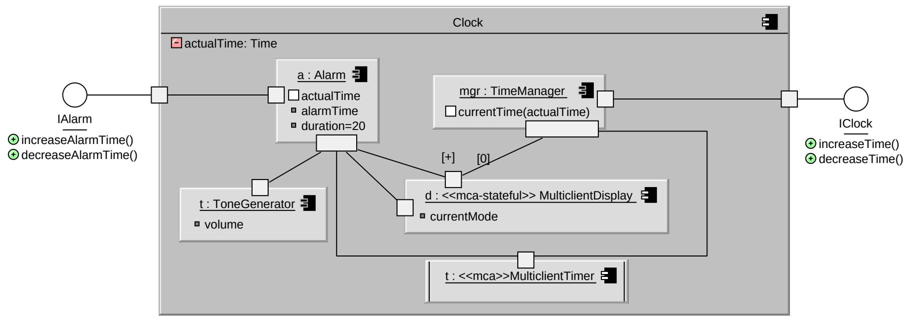
In order to solve the first problem, the Timer was replaced by a new composite called MulticlientTimer which conforms to the multi-client adapter pattern. Turning a single client component into a multi-client alternative is a common theme in the extension of architectures. This will be revisited in 9.3 when the display must be evolved to handle both the alarm and time display.

To solve the second problem, currentTime in TimeManager was made to be an environment property. This pushes the definition of the property "up to the environment" where it can then be aliased by the currentTime property of HourlyBeeper. Another common problem encountered is that information is localised and needs to be made available to another part of the architecture. Environment variables are one basic way of achieving this dispersal.

9.3 Adding an alarm

The alarm extension builds on the base architecture and adds an alarm which makes a continuous tone for a fixed duration of 20 minutes.

Figure 8: Adding an alarm to the base architecture (alarm)

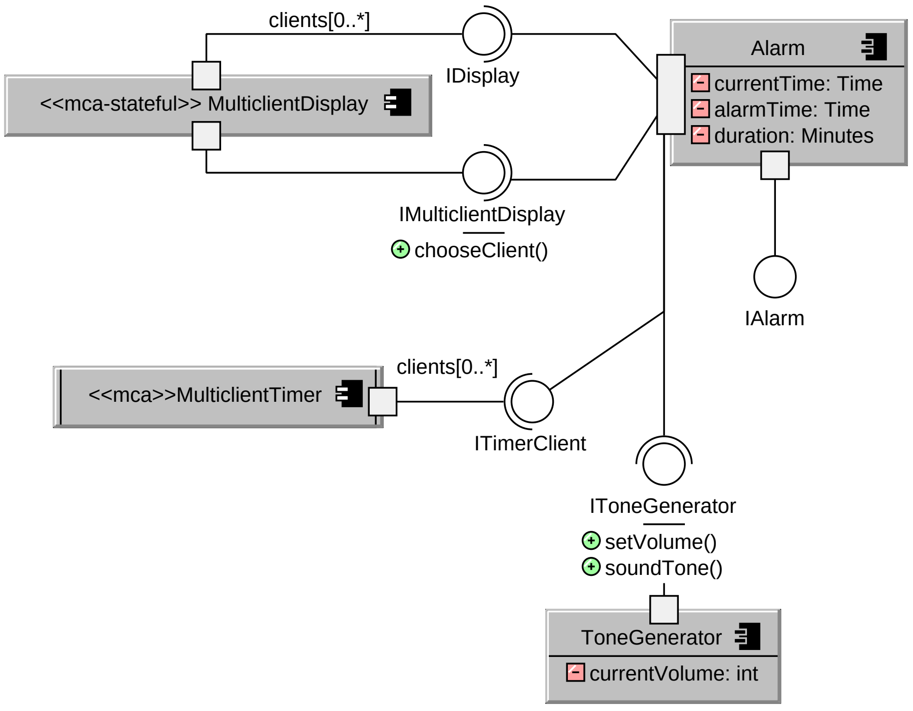


This extension faces the same problem as the beep extension: the Timer supports only one client. Again, sensible usage of the multiple-client adapter pattern solves the problem. In addition Display must be adapted to support multiple display clients. Unlike the timer, though, the display needs a stateful adapter.

Another problem shared with the beep extension is that Alarm requires the current time. This extension has also turned the current time into an environment property, but has chosen a different name. Although this is a contrived situation, even a simple name difference can make it difficult to integrate two sets of independently developed components [hoezle ref].

The alarm also needs to make an audible sound. This is accomplished through the ToneGenerator component which is further described below. The use of ToneGenerator here and Audio in the beep extension indicates another issue with independent development: duplication of effort to achieve the same goal. In this case, Audio offers a strict subset of the features of ToneGenerator.

Figure 9: The internal components of the alarm clock



9.4 Progressing the base: adjusting the time from a radio signal

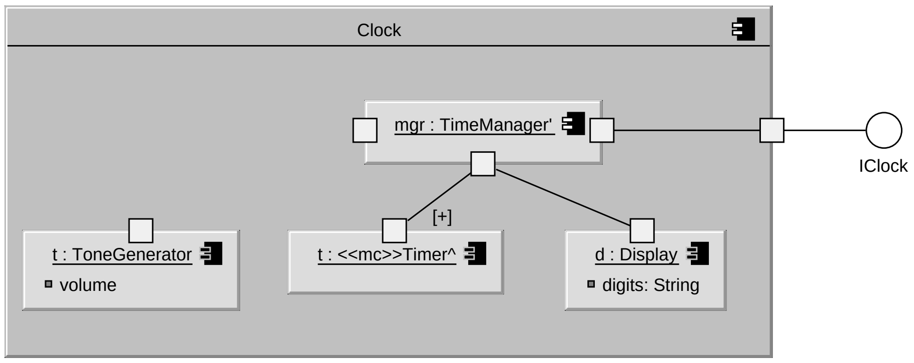
The clock1.1 stratum represents an extension that progresses the architecture. The maintainers of this architecture reviewed some of the extensions and decided to rework the architecture to be more extensible. In addition, it was decided to synchronise the time with a radio signal, as available in some European countries. The changes made to the architecture are:

1. Integrate the ToneGenerator directly into the architecture as a base component, even though it is not connected by default.
2. Upversioned the Timer to support multiple clients.
3. Upgraded Timer to only support the IFineTimer interface, which notifies each tenth of a second (not a subtype of previous interface).
4. Added an interface ITimeControl on the TimeManager, to allow setting and getting the time.
5. Add the radio signal (RF) synchronisation facility.

Items 1, 2 and 3 illustrate a common process: when evolving a base architecture, the developers may decide new features based on issues or requirements that existing extensions have had to deal with. This feedback cycle produces a strong evolution path for the architecture, but has the undesirable effect of requiring rework of existing extensions. In many systems, this process acts as a significant deterrent for existing systems to move onto an evolved architecture.

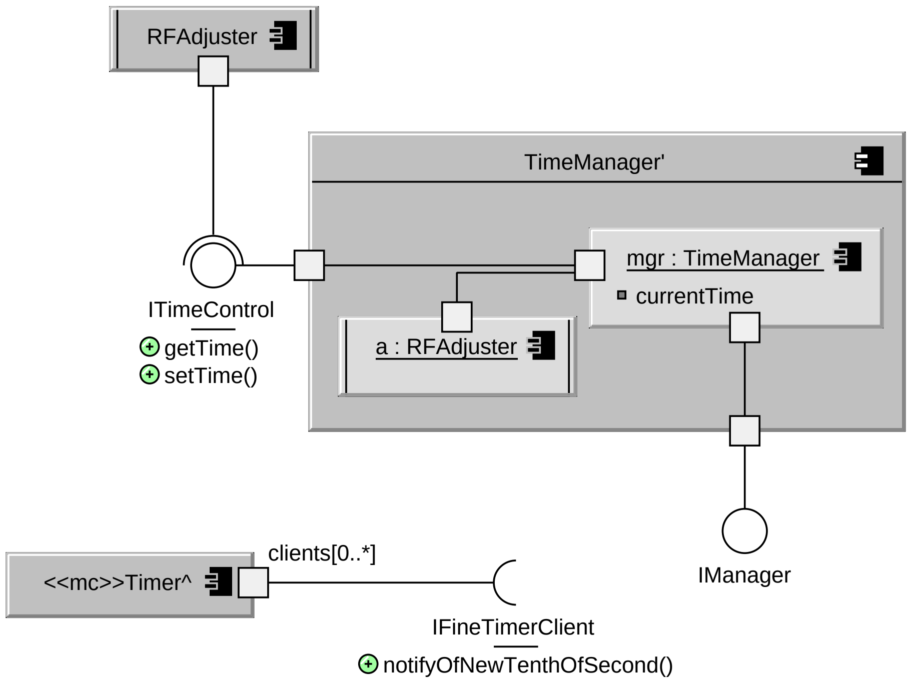
For added readability, the upversioned leaf components are shown with a ˆ after their name. Supplanting is shown by a ' after the name.

Figure 10: The progressed architecture (clock1.1)



The internal component definitions are shown below. The upversioned Timer is marked with the «mca-added» stereotype to indicate that it has added multiple-client capabilities, and supersedes any MulticlientTimer components.

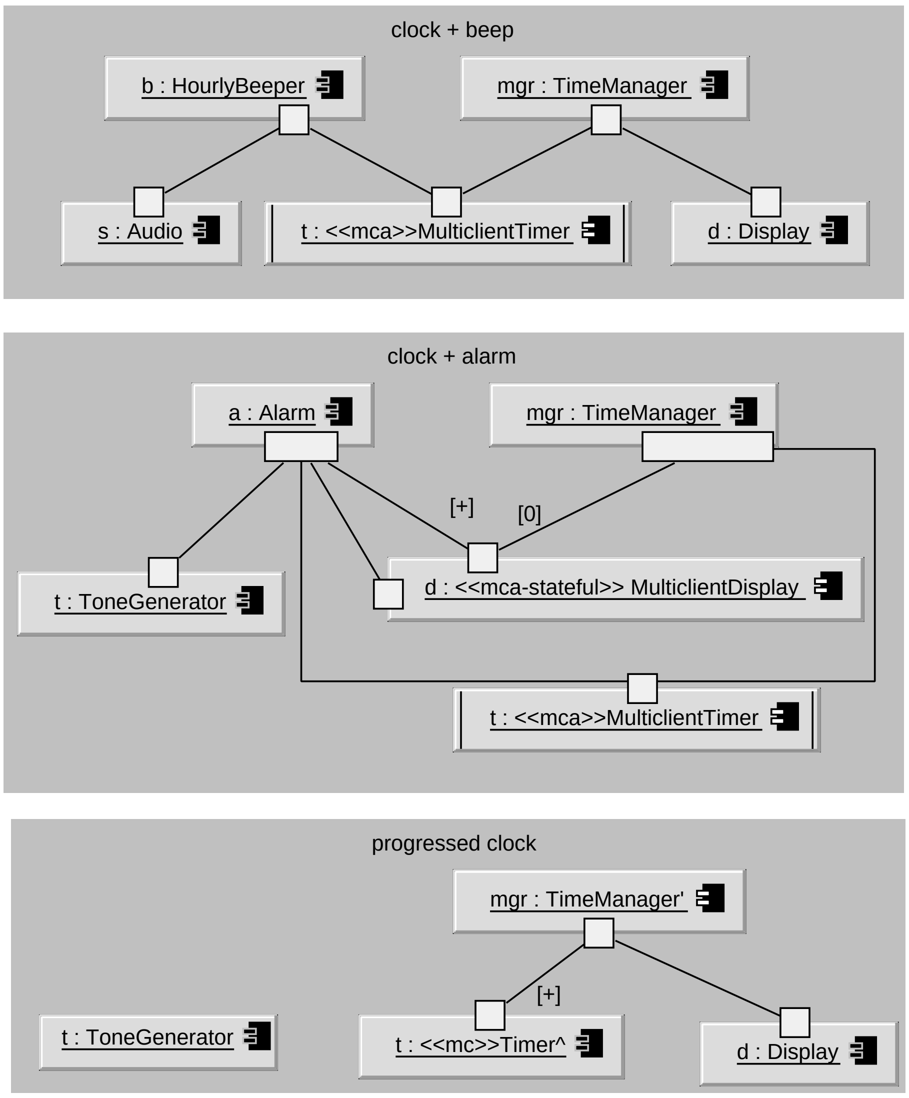
Figure 11: The internal components of the progressed architecture



9.5 Combining the extensions

Applying the extensions to the base architecture and flattening to remove composites reveals the discord between the three scenarios. Although they are not dissimilar, the differences are enough to cause major integration problems.

Figure 12: The internal parts of the three extensions



The combination rules progress as follows:

1. Copy each component in the progressed stratum into the new stratum.
2. For any extensions:
   (a) Replace any «mca» or «mca-stateful» components with possible progressed counterparts.
   (b) Merge the internals