Runtime Mixn and Match Design Pattern

Paul G. Austrem
Dept. of Information Science and Media Studies
University of Bergen
Fosswinckels gt 6, 5007 Bergen
0047 55 58 41 18
paul.austrem@infomedia.uib.no

ABSTRACT
Ubiquitous computing is becoming reality, vendors are introducing products that support ubiquitous entertainment and media solutions, businesses are adopting service-oriented architectures, and mobile devices are becoming service consumers. To accommodate this foundational change, software needs to be dynamic and adaptable. This work proposes a pattern for resolving the need for dynamic actors by introducing the concepts of Intents, IntentHandlers, IntentFilters and IntentResponders. These four concepts express an abstraction allowing for late dynamic runtime binding to solve functional exigencies. Client software is no longer bound to specific programs, functions or services to solve functional needs; instead they can dynamically bind to IntentResponders to solve their functional exigencies. The pattern may incur a slight performance overhead, but allows for an extendable and dynamic solution.

Categories and Subject Descriptors
D.2.11 [Patterns]: Composite pattern – late binding, decoupling, distributed solutions.

General Terms
Design

Keywords
Strategy pattern, runtime binding, architecture

1. INTRODUCTION
The ongoing movement towards the mobile and ubiquitous computing is beginning to have implications on the way we design software. The notion of ubiquity, and the concept of what has become known as "cloud computing" [1-3] promotes fresh requirements at the architectural level when developing software. The term "cloud computing" is at present still a bit fluffy, and depending on whom you ask the definition may vary.

This work defines cloud computing as meaning systems strongly reminiscent of Service-Oriented Architecture based systems, wherein clients (a client can be another service or an end-user), also known as service consumers, can use published services to fulfill functional needs (a service is a concept, and can be an internal function, external application or an external data provider). This imposes the need for software to be adaptable and dynamic, beyond being just extendible and maintainable. By saying adaptable and dynamic we mean systems that can dynamically adapt to resolve functional challenges. A client could, given a specific functional need, (hereafter referred to as an intent) dynamically bind to a service at runtime to fulfill the intent.

These novel needs will likely lead to a change in the foundational architecture of systems. This work presents a pattern that can aid in achieving this through the use of intents and resolvers. The pattern is an architectural pattern, and is currently utilized in for example the Google Android platform [4].

The following section will introduce some of the background material motivating this pattern, this will be followed by the pattern "RUNTIME MIX'N MATCH".

2. BACKGROUND
Since the emergence of the object-oriented paradigm developers have tried to develop software that is extendable and maintainable, however there is reason to believe that the quality of autonomous should be added to this. The era of ubiquitous computing will require that our software is adaptable and can respond to challenges in its environment. The division between installed applications running on a device and applications leveraged as web services will be erased. Much work has been done in the domain of service discovery especially in the area of autonomous web service clients with dynamic discovery and binding [5-7]. This is not a new concept; however this pattern provides the essential architectural building blocks required to allow for runtime switching between different applications or services in order to fulfill a functional intent.

3. RUNTIME MIX'N MATCH

3.1 Intent
The Runtime Mix’n Match pattern allows for automated late binding of interchangeable services to fulfill intents.
3.2 Motivation
Suppose we are building a system on a mobile device, which allows us to edit an image taken with the device’s built-in camera and to post this image to an FTP-server. There are three applications on the mobile device that can assist us in achieving this, 1) is a pure image editor, 2) is an FTP-application, and 3) is an image editor with a built-in FTP-saving function. The only caveat is that the third, and most appropriate option, can only be run when the device is connected to the internet via a WLAN connection. Currently, our user is not in an area with a WLAN connection available. So the other two applications have to be used to solve the functional intent.

The user doesn’t wish to know about how the data is uploaded to the FTP-server, (s)he only needs to know that after the editing is complete, and (s)he presses the upload button, the file will be uploaded. The user’s intent is to upload a file, but how this is functionally achieved is of no concern to him/her. An intent is thus a desire to achieve a given functional goal, without having beforehand knowledge of how it is achieved. This can be achieved at the software level through the use of intents, IntentFilters, and IntentResponders. The concept of intents is central in this pattern – the notion of wishing to achieve a functional goal without specifying exactly how it is to be achieved. At the software level we are introducing a level of abstraction between the desire, or intent to do something and the actual function of doing it.  

3.3 Applicability
The pattern could be applied whenever you (as a client) are faced with a situation in which a part of your application may need to use an external application or service to achieve a functional intent (e.g. open a webpage, upload a file via FTP, present an image gallery). The specifics of how a functional intent is achieved, that is which service/application was used, is forgotten at the end of the session (a session in this respect could be the end of the photo taking, or when closing a file – essentially the work activity has been concluded). The main artifact of knowledge that is preserved is the intent, the desire to achieve a goal. Briefly stated, the desired end result is remembered, but the means used to reach it are discarded. There is a variation on this however, wherein both the end and the means are stored, and until specified otherwise the same means will be used in the following sessions [8, 9].

Avoid hard-coding how the functional intent should be resolved, for instance avoid direct calls to specific applications in your code. Instead use an intent, and let the IntentHandler decide how to resolve the intent. The applicability of RUNTIME MIX’N MATCH should be apparent if you have software that is strongly coupled to a certain external service or application in order for it to correctly execute.

Another scenario in which the pattern could be applied is within the context of service-oriented architectures. The pattern would then represent a significant infrastructural element. Essentially, the entire messaging architecture would be based on the pattern, and its concepts of Intents, IntentHandler, IntentFilters and IntentResponders.

3.4 Participants
The following classes are the most central in the pattern.

- **Client**
  - in this pattern is any element (person or application) that has a functional intent that needs to resolve, but it does not care about how this intent is resolved functionally. It merely issues its intent to the IntentHandler and expects the intent to be resolved.

- **Intent**
  - declares the structural attributes required in order to submit an intent and get a satisfactory reply. Contains an attribute of type IntentResponder so it can keep a reference to any suitable ExternalApplication to resolve that specific intent. For instance if you consider streaming video, the first time around the Client will submit an Intent and receive a reference to an ExternalApplication (in the form of an object of type IntentResponder) which can then be used at a later occasion of video streaming.

- **IntentHandler**
  - the IntentHandler receives the Intent from the Client. Its task is to enquire as to whether there are any applications that have IntentFilters that match the attributes of the Intent of the Client. If a match is found, then this will be passed back to the Client from the IntentHandler.

- **IntentFilter**
  - The IntentFilter class contains some of the same attributes as the Intent class, and similarly to the Intent class its task is to function as a structural class holding the three attributes that an Intent is matched against, namely action, type and component. It will also hold a reference to the ExternalApplication through an IntentResolver interface type, so that the IntentHandler can pass this back to the Client.

- **IntentResponder**
  - this is, as previously mentioned, a marker interface, thus it declares no methods. It is used by the Client and the IntentFilters to check that any potential ExternalApplication actually does support intent resolution, and also allows the Client to bind to different IntentResponders at run-time.

3.5 Structure
The structure of the pattern contains the STRATEGY pattern [10], thus it is a composite pattern. The Client class represents any application or person with a desire to accomplish a functional task, or more precisely in this nomenclature, it has a functional intent. Thus, the Client may maintain an association to a Intents. A Client may generate many intents during its execution that are to be for example sequentially executed. Note however that these intents are “personal” to the Client, thus intents cannot be shared

---

2 Note that the concept of "marker interfaces" is used in Sun’s Java; for instance when marking a class as serializable, or clonable by implementing the marker interfaces of Interface Serializable in the package java.io and Interface Clonable in the package java.lang respectively.
between different Clients. This leads us to the abstract class Intent and its derivative(s). The Intent class is essentially just a structural class containing the constitutive attributes required to express an intent. The attribute data contains the information about what data you wish to edit or retrieve, for instance if you wish to edit an image stored on your mobile device, then the attribute data would contain the path to the appropriate file, or if the image is treated in memory then it would contain a binary datastream.

The other attribute, action, contains information about the action that is to be performed. These actions could be expressed in the form of an enumerated list, for instance EDIT_ACTION, VIEW_ACTION, OPEN_ACTION, DELETE_ACTION, ONLINE_ACTION, etc.

The three remaining attributes are not equally integral, for instance the attribute type can hold a description of the data/file type that is involved in the action, for instance it could hold "text/html" for a webpage, "audio/wav" for an audio file or "text/plain" for a generic text document. However this is not necessarily necessary, because this information could be inferred from the attribute data (but by all practical means the use of type is encouraged to avoid unsightly string parsing when inferring the type through data). The fourth attribute in the Intent class is component. This attribute is concerned with explicitly denoting a specific component to be used in order to resolve the intent, for example if you wish to use a specific component to fulfill your intent, maybe because the component is signed, or verified, or you beforehand know that it is well suited (for example a compression algorithm), then this can be declared in the component attribute. The last attribute, extra is used to package additional payload information, for instance if your intent is to watch a streamed video then the extra attribute could contain the stream data. Finally, the Intent class has an attribute intent_responder of type IntentResponder (a marker interface which will be discussed later).

The Intent class is an abstract class, thus variations on intents can be added to the client without inducing changes to the client code. The Client class is also associated with the class IntentHandler, this associated class is the manager of all received intents. Note that the class IntentHandler could potentially be a static class, ensuring only one instance of it (alternatively the SINGLETON pattern could be used although neither has been done in this approach).

When a Client issues an Intent this is passed parametrically to the IntentHandler. The IntentHandler will based on the contents of the attributes data (or preferably type) and action (and possibly the other attributes if they have been set), lookup whether there are any IntentResponders that match the criteria set in the Intent object’s attributes. This is accomplished by mapping the attributes to the IntentFilters of an ExternalApplication implementing the marker interface IntentResponder. This is the reason the IntentResponder marker interface is included, in order to signal that an application does offer intent resolution given the matching of it’s intent.filters. However a more sustainable solution when dealing with proprietary External Applications where you do not have preconditioned entry points for invocation would be to use the Marker Interface as a fully operationalized interface.

3 In Java 1.5 this could be solved without the use of a marker interface by using annotations to denote that certain applications offer intent resolution. In C# it could be done with attributes.
Note also that there exists a strong composition relation between the IntentFilter class and the ExternalApplication class, this is because an IntentFilter cannot exist without it being associated with an ExternalApplication. Once again the marker interface IntentResponder makes its mark. This is due to the fact that an IntentFilter can only be associated with an ExternalApplication which implements IntentResponder.

3.6 Collaborations
Although some of the collaborations and behavior has been outlined above in the sections Structure on page 2 and Participants on page 2, this section will represent this in the form of a sequence diagram, delineating all the operations and messages that are involved in the whole process from Intent creation to resolution.

A client will initiate the process through creating an Intent object. The Intent object’s attributes data, action, etc. will be when it (an Intent) is initialized. When this is completed, the Client object a_client will parametrically pass the an Intent object to the instance a_handler of type IntentHandler by calling the method match_intent(...), which will invoke a_handler’s comparison method (in this work, see section Sample Code on page 5, the comparison has been handled through operator overloading).

This method will iterate through all IntentFilters and when an application with a matching IntentFilter is found this will be returned to the Client a_client as an object of the type IntentResponder. Finally the Client a_client will set the intent_responder attribute of the Intent an Intent object. When this is done, the Client a_client will have a reference to an external application which can be invoked whenever, in the future as well, a need to resolve the same intent arises (for instance video streaming, playing an audio mp3 file, etc.). The resolved intent could moreover be serialized and reconstructed the next time the application is started, thus allowing for a permanent intent resolution.

3.7 Consequences
The following consequences have been identified and should be considered when applying this pattern:
A list of selected benefits and liabilities follows, however note that some of these may be mitigated through the implementation.

Benefits:
- **Increases flexibility** (at the cost of complexity and performance). The pattern removes the need to strongly bind any activity (playing a video, sending an email) to a specific application. Instead this can be handled at runtime. The cost of this is the increased complexity of the design, and a potential reduction in performance (due to increased messaging, pass-by-value, matching intent against intent filters).
- **Reduced coupling.** The pattern encourages looser coupling between software components in terms of “intent handler” and “clients” separating the functional resolution from the invoker.

Liabilities:
• **Increased messaging and use of reflection.** Because the pattern relies so heavily on late/runtime binding, this will cause increased use of reflection. For instance when checking against the marker interface `IntentResponder`, this could require checking whether the `ExternalApplication` object implements the `IntentResponder` interface. Additionally, the pattern is chatty; many messages are exchanged between the `Client`, `IntentHandler`, `IntentFilter` and `IntentResponder`. Performance-wise, using `instanceof` (Java), `dynamic cast` (C++) or `is` (C#) to check whether an `ExternalApplication` implements the `IntentResponder` interface does not give a heavy performance hit, similarly all the messaging will not noticeably affect performance. However if the pattern is applied in a networked scenario any network latency or congestion could affect the perceived performance. The use of the marker interface is, as mentioned, necessary in order to facilitate late/runtime binding of intent resolvers to clients.

• **Increased complexity.** Since the pattern does introduce new classes, and uses delegation and abstraction to achieve the runtime matching and binding, this will increase the complexity of the system.

### 3.8 Implementation

The pattern may utilize different models to handle certain parts of the process. For instance, in this work the matching process is performed by the `IntentHandler` as it maintains a registry of all `IntentFilters`. Each `IntentFilter` is associated with only one `ExternalApplication`. The `IntentHandler` iterates through all the `IntentFilters` and returns the associated `ExternalApplication` as a type `IntentResponder` of any `IntentFilters` that match the original `Intent`.

However, another approach could be to associate the `IntentHandler` directly with the `ExternalApplications`; this could allow the `IntentHandler` to check if an `ExternalApplication` implements the `IntentResponder` interface, in which case it could lookup its `IntentFilter` and see if it matches. The advantage of this model is it could be used in a more introspective approach. The `IntentHandler` could thus check with newly added plugins/applications whether they implement the `IntentResponder` interface, in which case they can expect to find (an) associated `IntentFilter(s)`. This approach relieves the `ExternalApplication` from having to register its `IntentFilter(s)` with the `IntentHandler` upfront. Instead it can all be handled at runtime.

If implemented in a service-oriented architecture context, it could be viable to place the `IntentHandler` on a separate server, thus all the clients utilizing a `Client’s Intents` would be submitted to the server, ensuring a centralized handler for all registered `IntentFilters`.

The marker interface `IntentResponder` can be fully operationalized to also provide entry points for invoking the `ExternalApplications`. This is for instance done in COM’s `IDispatch` [11], where the interface is operationalized with methods that allows remote invocation of object’s implementing it. In a heterogeneous environment this could be a platform independent solution, whereas if you have full control over the interfaces of all the external applications you could make do with a marker interface. Frequently however this will not be the case, and in which case you would need to operationalize the interface (using for example `IUnknown` or `IDispatch [11]`) so that all `ExternalApplications` do provide a method for querying their interfaces.

### 3.9 Sample Code

The following code is written in C# using some of the idioms of the language (such as generics and the C# take on Enums). However the code is still representable as high-level code providing an understanding of how the pattern can be implemented.

```csharp
class Client
{
    //conceptually equivalent of the Client*/
    static void Main(string[] args)
    {
        IntentHandler handler = new IntentHandler();
        //register two IntentFilters with two external applications*/
        IntentResponder externalApp;
        IntentFilter filter;
        externalApp = new externalApplication("Notepad");
        filter = new IntentFilter((int)Utility.Actions.Edit, null,
                                 "text/plain", externalApp);
        handler.add_filter(filter);
        externalApp = new ExternalApplication("Wordmate");
        filter = new IntentFilter((int)Utility.Actions.Edit, null,
                                 "text/plain", externalApp);
        handler.add_filter(filter);
        //create concrete intent, and hand it over to the handler*/
        Intent myintent = new
        ConcreteIntent((int)Utility.Actions.Edit,
                       "d:\mytext.txt", "text/plain", null, null);
        //place the results in a List of viable IntentResenders, if any exist*/
        List<IntentResponder> matching_responders =
            handler.match_intent(myintent);
        //check for results, if any responders are found,
        //print their names to the screen*/
        Console.WriteLine("Found "+
                          matching_responders.Count.ToString() + " matching 
                          IntentResponders");
        for(int j=0; j<matching_responders.Count; j++)
        {
            ExternalApplication e =
                (ExternalApplication)matching_responders[j];
            Console.WriteLine(e.ProgID);
        }
    }
}
```

Listing 1. Code for the Client class

The `Client` is here bundled together with the entry point of the sample code, thus it is entangled in the creation of the `IntentHandler` and some `IntentFilters`.

```csharp
abstract class Intent
```
the abstract class Intent with its associated attributes and constructor*/
public int action
public string data
public string type
public string component
public string[] extras;
private IntentResponder intent_responder;

public Intent(int p_action, string p_data, string p_type, string p_component, string[] p_extras )
{
...
}

class ConcreteIntent : Intent
{
public ConcreteIntent( int p_action, string p_data, string p_type, string p_component, string[] p_extras) : base(p_action,p_data, p_type, p_component, p_extras)

Listing 2. Code for the Intent and ConcreteIntent classes

Listing 2 shows the core code for the structural classes Intent and ConcreteIntent. Note that the ConcreteIntent class’ constructor merely delegates the whole process to the super constructor (in C# this is done by a call to “base:”, whereas in Java the equivalent would be “super()”).

class IntentHandler
{
    private List<IntentFilter> registered_filters;
    private List<IntentResponder> matching_responders;
    public IntentHandler()
    {
        registered_filters = new List<IntentFilter>();
    }
    public void add_filter(IntentFilter filter)
    {
        registered_filters.Add(filter);
    }
    public List<IntentResponder> match_intent(Intent an_intent)
    {
        /*lazy initialization*/
        if (matching_responders == null)
        {
            matching_responders = new List<IntentResponder>();
        }
        matching_responders.Clear();
        /*clear it of previous results before adding matching responders*/
        /*loop through all registered_filters and see which ones match*/
        for (int i = 0; i < registered_filters.Count; i++)
        {
            /*if a match is found, add it to the array*/
            if (an_intent == registered_filters[i])
            {
                matching_responders.Add(registered_filters[i].Associated_responder);
            }
        } /*return the array upon completion*/
        return matching_responders;
    }
}

Listing 3. Code for the IntentHandler

In the above Listing 3 we see the code for the IntentHandler, note that in this sample we have applied operator overloading for the relational operator == (marked in yellow), the actual overloading is shown in Listing 4. All matching IntentResponders are placed in an array and returned to the Client upon completion.

In Listing 4 below we can see that a few C# idioms are applied in the use of generics and the pairwise operator overloading (both == and != are overloaded). Depending on the implementation language, and whether one chooses to use operator overloading to achieve the desired effect of comparing IntentFilters with Intents, the IntentHandler class could, like the Intent class, be a purely structural class.

class IntentFilter
{
    public int action;
    public string component;
    public string type;
    public IntentResponder associated_responder;

    public IntentFilter( int p_action, string p_component, string p_type, IntentResponder p_associated_responder )
    {
    ...
    }

    public static bool operator ==( Intent clientFilter, IntentFilter registeredFilter )
    {
        /*overload the relational operator == to check Intent objects and IntentFilter objects*/
        if (clientFilter.Action == registeredFilter.Action &&
            clientFilter.Type == registeredFilter.Type)
        return true;
        else
        return false;
    }

    /*dummy implementation of != relational operator due to C# enforcement of pairwise overloading*/
    public static bool operator !=( Intent clientFilter, IntentFilter registeredFilter )
    {
        return false;
    }
}

Listing 4. Code for the IntentFilter class

The final listing, Listing 5, shows the class ExternalApplication and the marker interface IntentResponder. Further descriptions of the role of the marker interface is not needed, note that the ExternalApplication class does provide an attribute progID. This is merely used in the code to differentiate between IntentResponders.

class ExternalApplication : IntentResponder
{
    public string progID

    public ExternalApplication( string p_progID )
    {
    ...
    }

    /*the marker interface – which could of course be operationalized, e.g. as in IDispatch*/
    interface IntentResponder
    {    }
Listing 5. Code for the ExternalApplication implementing the marker interface

The output of the code sample above would yield:

![Found 2 matching IntentResponders Notepad WordPad](image)

Figure 3. Output from code sample

The only aspect missing from this simplified code sample is a mechanism allowing ExternalApplications to register their IntentFilters. If this is done at runtime the IntentHandler would need to supply an accessible method for registering an IntentFilter. However some implementations utilize a static approach by loading the information about IntentFilters from a serialized source, e.g. a flat file, or an XML file.

3.10 Known Uses

The pattern is applied by Google in their Android framework [4, 12] for development of software for mobile devices. In Android, it forms a substantial part of the core infrastructure and provides an abstraction allowing developers save time normally spent on resolving functional activities at compile time, by deferring it until runtime where decisions about how to handle an activity can be resolved through runtime binding. Practically in Android when developing a solution you can define an Intent as an abstract description of an operation to be performed. This intent can then be broadcast by using sendOrderedBroadcast() or sendStickyBroadcast() [13] to a registry of BroadcastReceiver simple. If any of the receivers are able to resolve the intent in the broadcast then they method. There are many variations on how the intent resolution can be handled in the Android framework. The interested reader is referred to [4, 12, 13].

The pattern is also used in the Windows XP operating system through the "OpenWith ProgIDs" and "OpenWithList" verbs [8], wherein it is possible to right-click a file and select "open with". This will generate a list of applications that may resolve your intent (to open a file of a specific type). Windows XP offers various verbs ("open", "edit", "play", "print", "preview") [9] to express the action of the Intent. Thus, the operating system functions as an IntentHandler, maintaining in its registry (under HKKEY_CLASSES_ROOT) information (the equivalent to an IntentFilter) about which applications are registered to handle the desired verb (referred to as the action attribute of the Intent in the pattern) for this filetype (referred to as the data attribute of the Intent in the pattern). Finally the IntentResponder is then selected from the list that appears in the "Open With" list, and the Intent is fulfilled.

3.11 Related Patterns

The pattern incorporates at its heart the essence of the STRATEGY pattern [10]. The STRATEGY design pattern encourages two important design principles; namely "program to an interface not implementations” and "encourage composition over inheritance” [14] (page 32). The intent of the STRATEGY pattern is "define a family of algorithms, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from clients that use it.” [10] (page 315), in its essence animates the two aforementioned principles. In runtime MIXN MATCH the same concept applies, but it is only in solutions where the “Marker Interface” is operationalized to a “full” interface that the similarity becomes obvious. When this is done the various ExternalApplications implementing the interface become the ConcreteStrategies [10].

The FACTORY METHOD pattern could be used when creating concrete Intents, because if it ever becomes necessary to add new concrete intent types (you could for instance have system intents that deal with low level system functions, then these could easily be added without inducing any change in the closed part of the design (the Intent, IntentHandler, Client).

The COMMAND pattern [10] could be used in the case of the IntentResponder to encapsulate the actual invocation of the ExternalApplication. The SINGLETON pattern [10], could be used to ensure there is only one instance of the IntentHandler.

4. ACKNOWLEDGEMENTS

I would like to thank my shepherd Bob Hanmer for his insights, contributions and encouragement during the work with this paper. Thanks also go to my supervisor Andreas Opdahl for his suggestions and comments to the early versions of this paper. Finally I would like to thank my fellow PLoP participants for their valuable and constructive feedback.

5. REFERENCES


