# Blink componentization after the Chromium-Blink repository merge

haraken@chromium.org, tasak@chromium.org

2015 Mar 20

**Summary:** This document proposes a way to reorganize Blink directories and their dependencies after the repository merge. If you are busy, you can just read the “Proposed architecture” section.

## Motivations

The Chromium repository and the Blink repository are going to be merged. core/ and modules/ in the Blink are going to be split into separate components. The current architecture that connects Chromium and Blink are broken in many ways (as described below).

Now it's time to consider a plan to make the architecture better. This document proposes a way to re-architect Blink directories and their dependencies after the repository merge and the core-modules componentization. The goal of the re-architecture is to remove unnecessary layering abstractions between Chromium and Blink and make the interaction *simpler* and *faster*.

This proposal is for ***Blink componentization v1***, the goal of which is to fix the architectural problems in Blink we're actually facing now. No substantial changes in content/ are planned in Blink componentization v1. ***Blink componentization v2*** may aim at a more radical re-architecture such as eliminating public APIs completely etc. I'll discuss Blink componentization v2 in the "Open questions" section, but the goal is still not clear to me.

It is important to understand that refactoring is just a supporting technology to help make things happen (see the Blink midnight train). Thus when working on refactoring, it is important to understand how important it is to make things happen and the relative priorities with other Blink projects. From this point of view, personally, I think that the Blink componentization v1 is important to fix the current architectural problems and is a step which will be anyway required in the future. The Blink componentization v2 is unclear at the moment.

## The current architecture

(Note: To understand the problems we need to solve, this section describes a couple of concrete problems of the current architecture. You can skip reading the section if you're not interested in the mess. I'd recommend you don't spend lots of time understanding the section.)

### Dependencies in the current architecture

Fig.1 illustrates the dependencies of the current architecture.

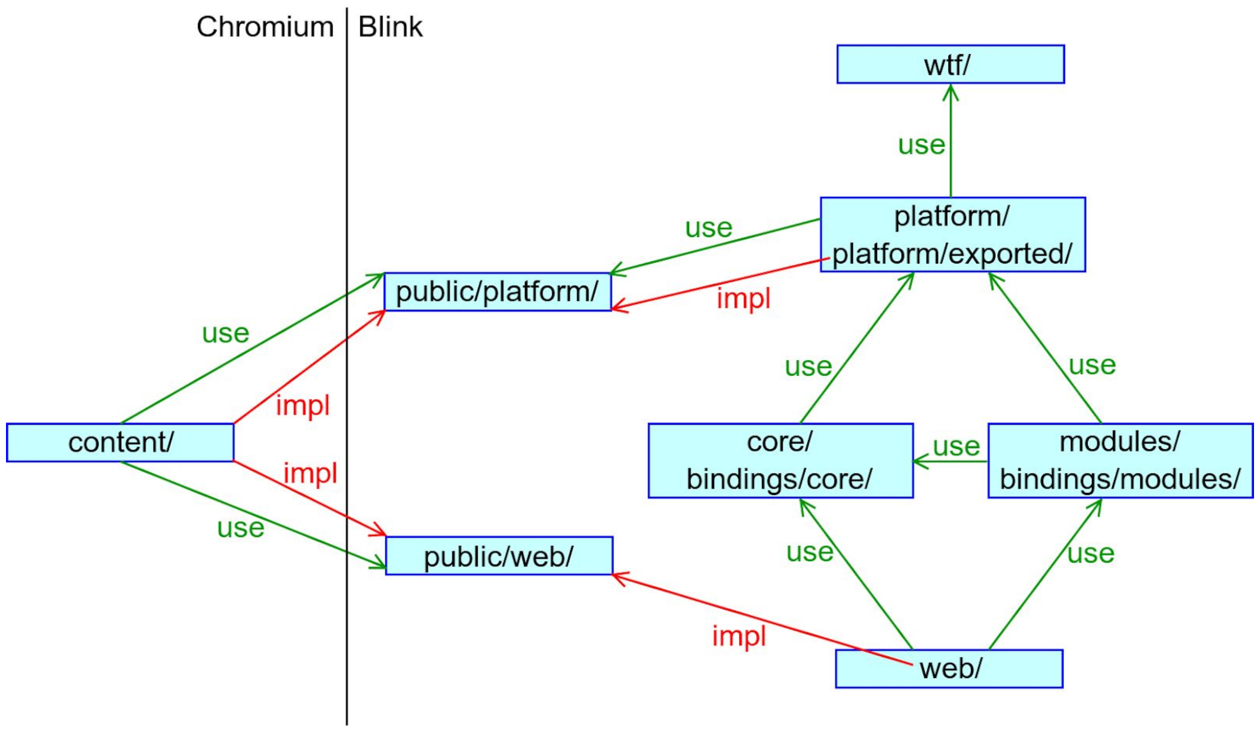

Fig.1 The current architecture

(Note: Figures in this document sometimes omit dependency arrows that can be transitively derived. For example, in Fig.1, a dependency arrow from web/ to wtf/ is omitted. A dependency arrow from web/ to public/platform/ is omitted etc.)

## WebDragData and DataObject

As a concrete example of the Chromium-Blink interaction, let's take a look at how `WebDragData` and `DataObject` work (Fig.2).

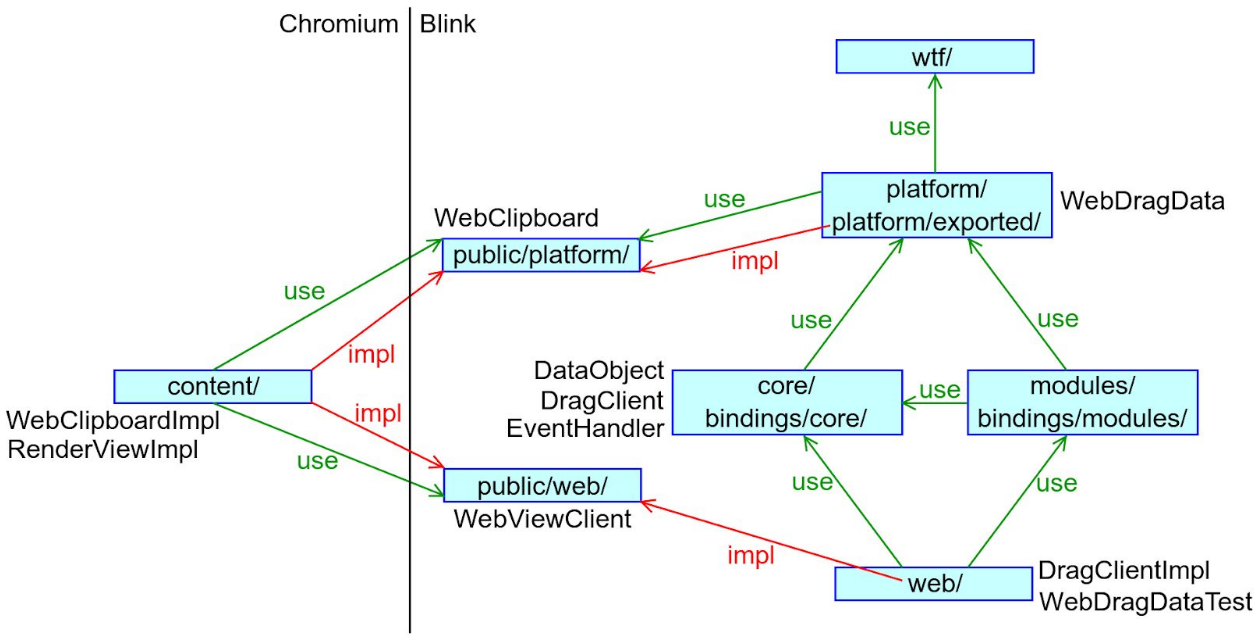

Fig.2 Relationships of classes around DataObject

First of all, `DataObject` is an object that represents data being dragged by a mouse device. `DataObject` is in core/.

```
// core/clipboard/DataObject.h
class DataObject {
  String m_filesystemId;
  Vector<DataObjectItem> m_itemList;
};
```

`WebDragData` is an object that wraps the `DataObject` to expose the data to content/. `WebDragData` is in platform/.

```
// public/platform/WebDragData.h
class WebDragData {
  WebVector<Item> m_itemList;
  WebString m_filesystemId;
};
```

Why is `WebDragData` in platform/, not in web/? This is because core/ needs to create a `WebDragData` object and call a callback method in content/. Specifically, core/clipboard/PasteBoard.cpp needs to call `Platform::current()->clipboard()->writeDataObject(WebDragData)`. Since core/ cannot use web/, we cannot put `WebDragData` in web/. Thus `WebDragData` is in platform/.

How is the `writeDataObject()` callback working? This is working by putting `WebClipboard` in platform/, putting `WebClipboardImpl` in content/ and making the `WebClipboardImpl` derive the `WebClipboard`. `Platform::current()->clipboard()` returns a pointer to the `WebClipboardImpl` and thus the `writeDataObject()` is redirected to the `WebClipboardImpl::writeDataObject()`. This is a common way to call content/ from Blink. In other words, we put an interface class (which is a collection of pure virtual methods) in platform/, put an implementation class in content/ and make the implementation class derive the interface class. Then Blink can call content/ through `Platform::current()->implementationObject()->callbackFunction()`.

That being said, things don't necessarily go so easy. That's why we have `DragClient` in core/ and `DragClientImpl` (which derives the `DragClient`) in web/. Basically, what we want to do here is just to call content/ from core/; we just want to let `EventHandler::tryStartDrag()` (in core/) call `RenderViewImpl::startDragging()` (in content/). However, we cannot simply use the above common pattern for the following reason. The `RenderViewImpl` derives `WebViewClient` in Blink. We cannot put the `WebViewClient` in platform/ because the `WebViewClient` uses a lot of things in core/. Thus we need to put the `WebViewClient` in web/. Then we somehow need to let `EventHandler::tryStartDrag()` call `WebViewClient::startDragging()`, but we cannot simply do this because core/ cannot call a method in web/. So we create the `DragClient` in core/ and the `DragClientImpl` (which derives the `DragClient`) in web/. Then `EventHandler::tryStartDrag()` (in core/) can call `DragClient::startDrag()` (in core/), which is redirected to `DragClientImpl::startDrag()` (in web/), which calls `WebViewClient::startDragging()` (in web/), which is redirected to `RenderViewImpl::startDragging()` (in content/). Well done.

Finally, why do we put `WebDragDataTest` in web/? Given that `WebDragData` is in platform/, it would make more sense to put `WebDragDataTest` in platform/. However, we cannot do that because `WebDragDataTest` needs to create a `DataObject` and the `DataObject` is in core/. Since platform/ cannot use core/, we cannot put `WebDragDataTest` in platform/. Thus `WebDragDataTest` is in web/.

Like this, things are very complicated. `DataObject` is just an example that has the complexity. Things around `WebIDBKey` are even more complicated. Also the core-modules componentization work by tasak@ has revealed that the current architecture has a lot of dependency violations. These dependency violations had been allowed (to some extent) in the pre-componentization world because core/, modules/ and web/ had been linked into one component. However, in the post-componentization world, the dependency violations are no longer allowed (it breaks builds). tasak@ has spent tons of time fixing the broken dependencies with introducing a bunch of abstraction layers.

In short, it is too complicated :)

Another problem of the current architecture is that we need to serialize and deserialize data between `DataObject` and `WebDataObject` every time Blink (or Chromium) passes the data to Chromium (or Blink). The serialization overhead wouldn't be a big deal in `DataObject` because mouse dragging won't happen so frequently, but can be a problem in performance sensitive objects such as IndexedDB. `WebIDBKey` is avoiding the serialization overhead by using an "inlining hack" (described in the next section) for performance reasons. Either way, the point is that we want to remove the abstraction layer and the serialization/deserialization overhead.

The important fact is that these problems come from the weird dependency constraints; i.e., core/ and modules/ cannot use web/, and platform/ cannot use core/ and modules/. The main motivation of the re-architecture is to simplify the dependency constraints and make the architecture simpler and faster.

## Oilpan and Vector

Next, let's take a look at a relationship between platform/ and wtf/. platform/ can use wtf/ but wtf/ cannot use platform/. This is because wtf/ and platform/ are linked into separate components.

Oilpan is in platform/heap/, and Vector is in wtf/. The reason we put Oilpan in platform/ is that Oilpan uses a couple of classes in platform/ (e.g., `WebThread`, `Task`, `ScriptForbiddenScope`, `TraceEvent` etc).

However, this causes a dependency problem. Oilpan provides a `HeapVector`, which is an on-heap version of a `Vector`. In other words, some part of wtf/Vector.h is implemented by Oilpan. This means that wtf/Vector.h needs to use methods in platform/heap/, but this is not allowed. This dependency issue is resolved by the following "inlining hack".

```
// platform/heap/Heap.h
class HeapAllocator {
  static inline void* allocateVectorBackings() {  // This is an
allocator for HeapVectors.
    …;
  }
};

// wtf/Vector.h
#include "wtf/DefaultAllocator.h"  // DefaultAllocator is an allocator
for normal Vectors. We don't need to include platform/heap/Heap.h.
template<typename Allocator = DefaultAllocator>  // We don't need to
write "HeapAllocator".
class Vector {
  Vector(size_t size) {
    Allocator::allocateVectorBackings(size);  // The above
HeapAllocator's method is inlined here.
```

```
    }
  };
```

The inlining hack is used in a lot of places in Oilpan and in Chromium-Blink interactions (e.g., `WebIDBKey`, `WebPrivatePtr` etc). It would be great if we could remove these hacks.

## Proposed architecture

I propose the architecture illustrated in Fig.3.

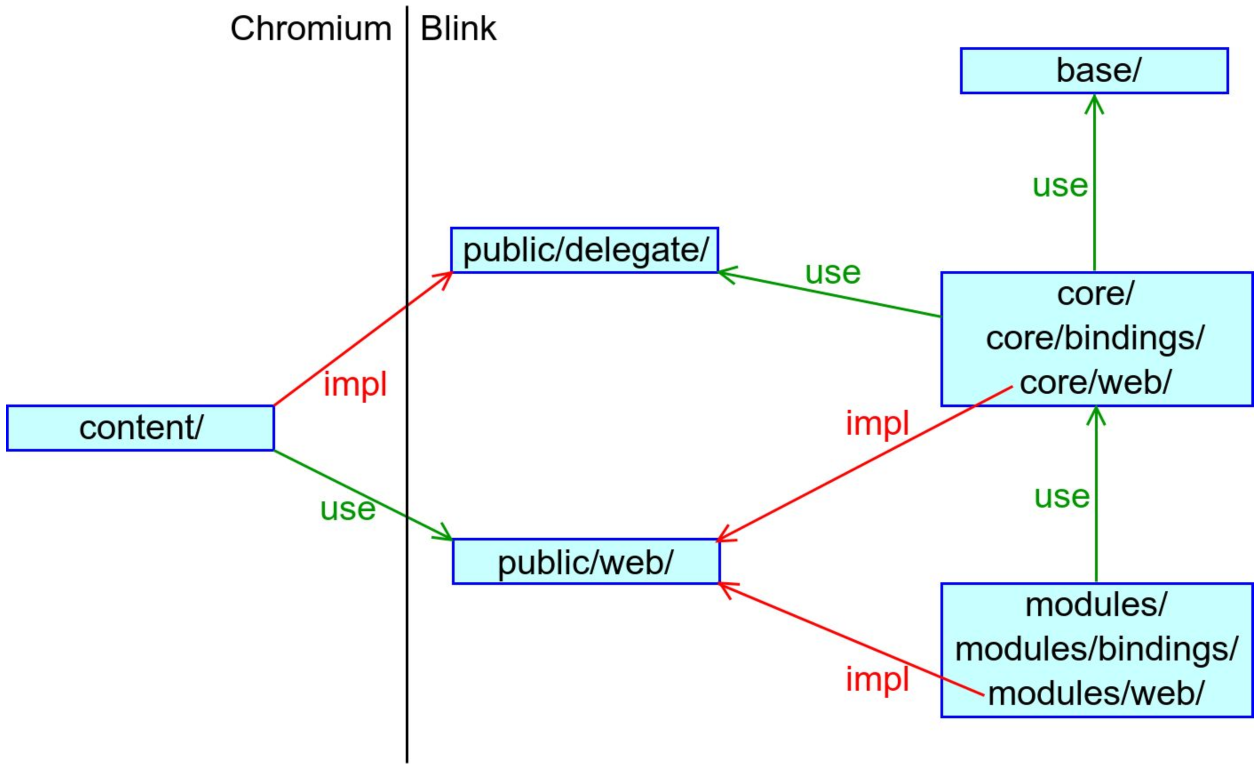

Fig.3 The proposed architecture

### public/web/

public/web/ contains WebXXX classes exposed from Blink to content/. For example, `WebNode` and `WebDragData` are put in public/web/. public/web/ contains only headers of the WebXXX classes and their implementations are put in core/web/ and modules/web/.  The WebXXX class holds a pointer to the corresponding XXX class via `WebPrivatePtr`.

```
  // public/web/WebNode.h
  class WebNode {
    WebNode firstChild();
    WebPrivatePtr<Node> m_node;  // WebPrivatePtr encapsulates
  RefPtrWillBePersistent<Node>.
  };

  // core/web/WebNode.cpp
  WebNode WebNode::firstChild() {
    return WebNode(m_node->firstChild());
  }
```

core/, core/web/ and core/bindings/ are linked into one component, so classes in these directories can touch any other classes in the directories. Similarly, modules/, modules/web/ and modules/bindings/ are linked into one component, so classes in these directories can touch any other classes in the directories. This is a key of the proposed architecture: you no longer need to introduce complicated abstraction classes to overcome the dependency constraints in the current architecture. Remember that the complexity in the current architecture is caused by the constraints that core/ and modules/ cannot use web/ and that platform/ cannot use core/ and modules/. These constraints are eliminated in the proposed architecture.

The content/ side can use the `WebNode` in the same way as it used to:

```
WebNode node = someNode->firstChild();
```

(Note: content/ normally uses WebXXX classes with stack-allocated objects.)

Just to clarify, the changes between the current architecture and the proposed architecture are as follows:

- WebXXX classes in public/web/ implemented by web/ will stay as is. Their implementations will be moved to core/web/ or modules/web/.
- WebXXX classes in public/platform/ implemented by platform/exported/ will be moved to public/web/. Their implementations will be moved to core/web/ or modules/web/.
- The current platform/ contains a lot of core-specific or module-specific code (for some reason). They will be moved to core/ or modules/.
- Abstraction classes that have existed just to overcome the current dependency constraints will be removed. In addition, unnecessary serializations/deserializations between XXX and WebXXX will be removed.
- No substantial changes will be needed in content/.

## public/delegate/

public/delegate/ contains WebXXX classes exposed from content/ to Blink. For example, `WebClipboard` is put in public/delegate/. public/delegate/ contains only headers of the WebXXX classes and their implementations are put in content/.

```
// public/delegate/WebClipboard.h
class WebClipboard {
  virtual void writeDataObject(const WebDragData&) = 0;
};

// content/renderer/webclipboard_impl.h
class WebClipboardImpl : public WebClipboard {
  virtual void writeDataObject(const blink::WebDragData& data) { ... }
};
```

Blink can use the `WebClipboard` in the same way as it used to:

```
Platform::current()->clipboard()->writeObject(webDragData);
```

(Note: You may wonder why we still need `Platform::current()`. In other words, you may wonder if Blink could just directly call methods in content/ once the repository merge is done. I'll discuss this issue in the next section.)

Just to clarify, the changes between the current architecture and the proposed architecture are as follows:

- WebXXX classes in public/platform/ implemented by content/ will be moved to public/delegate/. Their implementations will stay as is in content/.
- WebXXX classes in public/web/ implemented by content/ will be moved to public/delegate/.
- No substantial changes are needed in content/ or Blink.

(Note: I guess public/delegate/ is something that public/platform/ wanted to be. The current public/platform/ mixes classes implemented by content/ and classes implemented by Blink, but the original intention was to put only classes implemented by content/ into public/platform/. However, the original intention was not fulfilled because of the dependency constraints.)

## base/

base/ is a collection of Blink's basic architectures. The current wtf/ (e.g., `Vector`, `HashTable`, `String` etc) and the current platform/ except for platform/exported/ (e.g., `Oilpan`, `WebThread`, `Timer`, `Task`, `LifecycleObserver` etc) will be moved into base/.

By doing this, we can eliminate the dependency constraint that wtf/ cannot use platform/. We can remove abstractions classes and the "inlining hacks" that have existed to overcome the dependency constraint.

(Note: The point here is just that we want put wtf/ and platform/ into the same component. How to organize the directory structure in base/ is a separate discussion (e.g., base/wtf/ and base/platform/ etc).)

Just to clarify, the changes between the current architecture and the proposed architecture are as follows:

- wtf/ and platform/ except for platform/exported/ will be moved into base/.
- As a result, platform/ will be gone.

## Summary

In summary, the points of the proposed architecture are as follows:

- Remove the problematic dependency constraints by putting core/, core/web/ and core/bindings/ in one component, putting modules/, modules/web/ and modules/bindings/ in one component, and putting wtf/ and platform/ in one component (i.e., base/).
- Remove a bunch of abstraction classes and unnecessary data serializations/deserializations.
- Make the role of public/web/ and public/delegate/ clear. public/web/ is for classes exposed from Blink to content/, and public/delegate/ is for classes exposed from content/ to Blink.

Fig.4 illustrates the relationship between the current directory structure and the proposed directory structure.

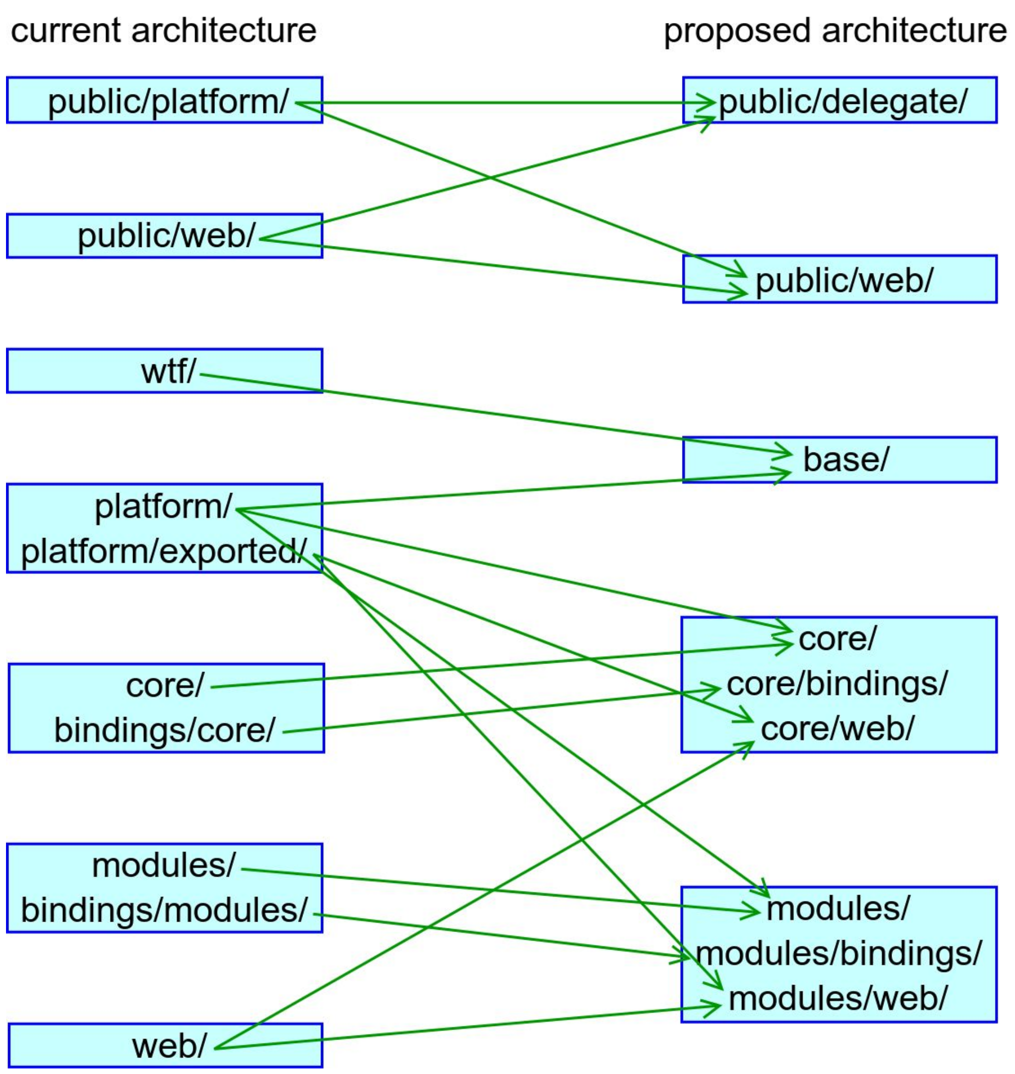

Fig.4 Directory migration to the proposed architecture

Fig.5 estimates the file counts in the current architecture and the proposed architecture. This is just a rough estimation and not that accurate. I think we can reduce the total number of file counts in the proposed architecture. Also the proposed architecture makes it easier to move things from core/ to modules/, so the file count balance between core/ and modules/ will improve.

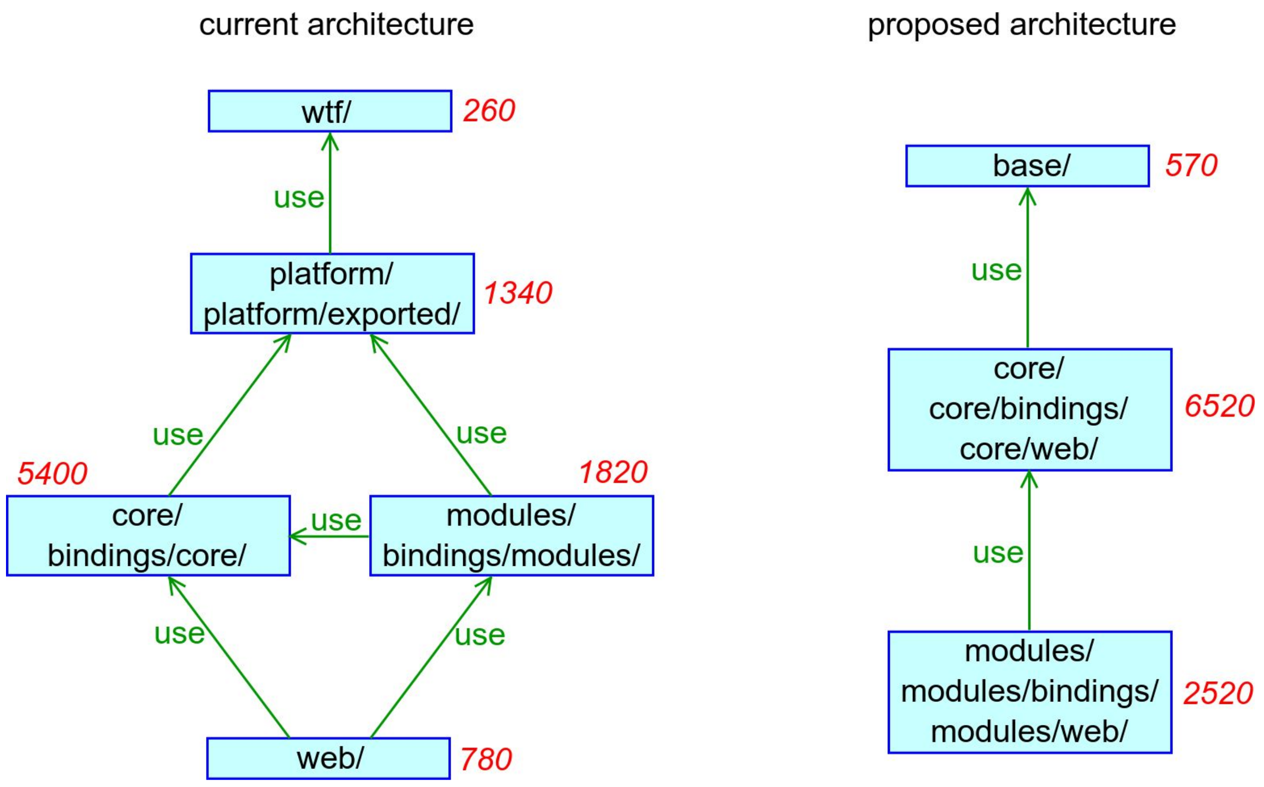

Fig.5. Estimated file counts in the current architecture and the proposed architecture

## Open questions

There are a lot of design issues. I'm open to ideas :)

### Another possible architecture

At first, I was considering an architecture illustrated in Fig.6.

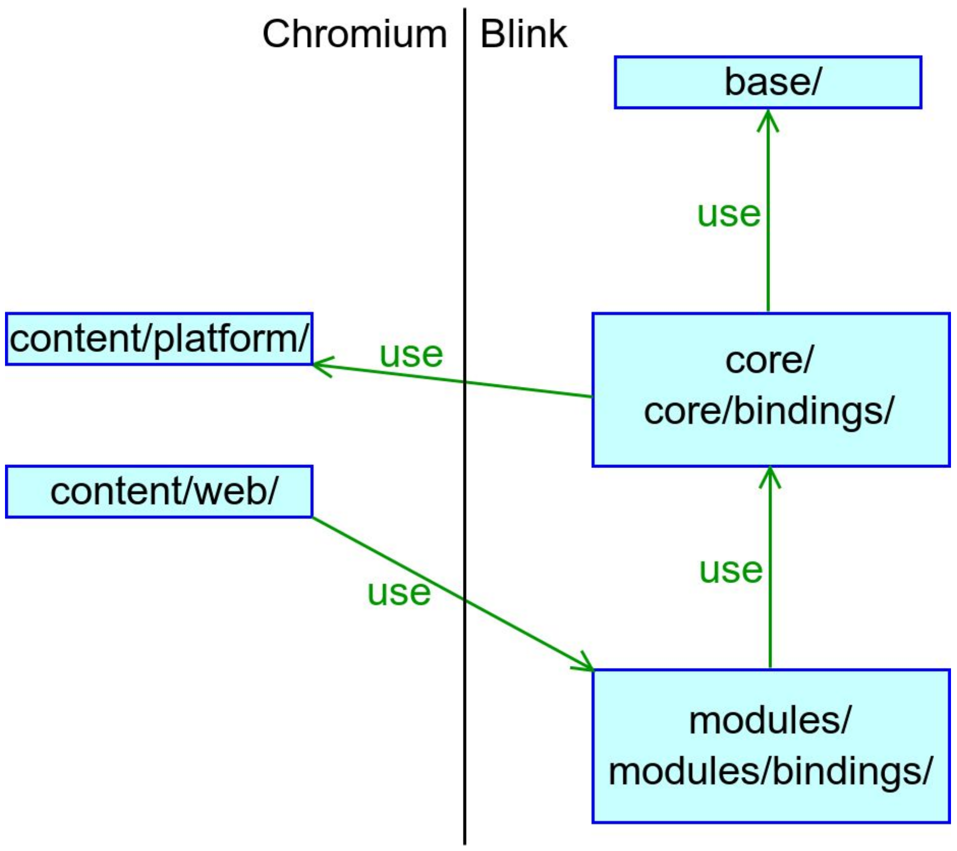

Fig.6 Another possible architecture

In Fig.6, content/ is split into content/platform/ and content/web/. In this architecture, content/web/ can directly call methods in Blink, and Blink can directly call methods in content/platform/. We no longer need `Platform::current()`. Blink is treated just as a part of content/ and this achieves a tighter integration between Chromium and Blink.

However, I realized that it would be hard to split content/ into content/platform/ and content/web/ like this. In other words, it would be hard to split content/ into one component that uses Blink but isn't used by Blink and the other component that doesn't use Blink but is used by Blink. Since the current content/ is written with the assumption that Blink is a "library" of content/, it is not easy (or/and it would not be a right thing) to reorganize content/ in that way.

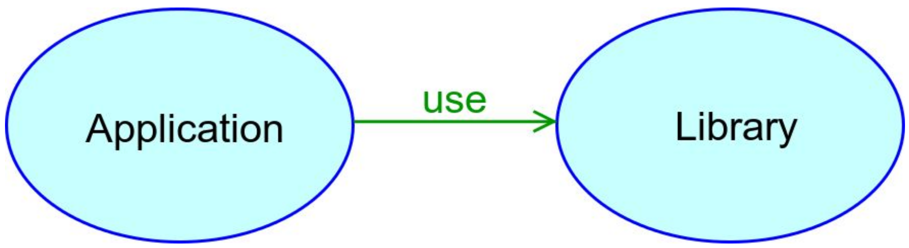

Fig.7 General dependency between an application and a library

In general, we should not add a dependency arrow from the “application” side to the “library” side (Fig.7). With the assumption that content/ is an “application” and Blink is a “library” (Note: I’m not sure if we want to keep this assumption or not), Fig.6 wouldn’t be a good architecture.

## How to implement WebXXX classes

In the proposed architecture, there is an inconsistency in a way in which WebXXX classes are implemented.

WebXXX classes in public/delegate/ are implemented by content/ with inheritance:

```
// public/delegate/WebClipboard.h
class WebClipboard {
  virtual void writeDataObject(const WebDragData&) = 0;
};

// content/renderer/webclipboard_impl.h
class WebClipboardImpl : public WebClipboard {
  virtual void writeDataObject(const blink::WebDragData& data) { ... }
};
```

On the other hand, WebXXX classes in public/web/ are implemented by core/web/ or modules/web/ with `WebPrivatePtr`:

```
// public/web/WebNode.h
class WebNode {
  WebNode firstChild();
  WebPrivatePtr<Node> m_node;  // WebPrivatePtr encapsulates
RefPtrWillBePersistent<Node>.
};

// core/web/WebNode.cpp
WebNode WebNode::firstChild() {
  return WebNode(m_node->firstChild());
}

// core/dom/Node.cpp
class Node {
  Node* firstChild() { return m_firstChild; }
};
```

The inconsistency comes from the following reason. If we implement WebXXX classes in public/web/ with inheritance, it will look like this:

```
// public/web/WebNode.h
class WebNode {
  virtual WebPtr<WebNode> webfirstChild() = 0;
};

// core/dom/Node.cpp
class Node : public WebNode {
  virtual WebPtr<WebNode> webFirstChild() {
    return WebPtr<WebNode>(m_firstChild);
  }
};
```

The inheritance model has the following issues:

- Inheriting from a class that has a virtual method adds a new vtable and increases sizeof(DOM object) by 8 byte. We don't want to increase the size of performance/memory-sensitive DOM objects.
- In this model, the content/ side needs to be rewritten as follows. This will require a substantial amount of work. I don't think the rewrite is worth doing.

```
// before
void foo(WebNode anotherNode) {
  WebNode node;
  node.appendChild(anotherNode);
  WebNode firstChild = node.firstChild();
  ...;
}

// after
void foo(WebPtr<WebNode> anotherNode) {
  WebPtr<WebNode> node = WebFactory::createNode();
  node->appendChild(anotherNode);
  WebPtr<WebNode> firstChild = node->firstChild();
  ...;
}
```

## Sharing more code between Chromium and Blink

One of the benefits of the repository merge is that it enables us to share more code between Chromium and Blink. Although the proposed architecture defines a clear border between Chromium and Blink (through public APIs in public/web/ and public/delegate/), once the repository merge is done, it is possible to eliminate the border and let Chromium and Blink interact more flexibly and share more code.

One example is Chromium's src/base/. src/base/ has a lot of things that can replace Blink's wtf/ (e.g., src/base/debug/, src/base/strings/, src/base/synchronization/, src/base/timer/, src/base/trace_event/ etc). I think we should do this. Blink should reuse more code in src/base/ and reduce code in wtf/. One way to do this would be to let Blink's base/ depend on Chromium's src/base/.

That being said, it would be too optimistic to consider that we can remove the majority of wtf/. Below are a couple of examples which would be hard to be replaced with src/base/ or std libraries:

- Blink's `Vector` wouldn't be easily replaced with `std::vector`. This is because Oilpan needs to implement an on-heap version of Blink's `Vector`. For the same reason, `HashTable`, `HashMap`, `ListHashSet`, `HashCountedSet`, `Deque` and other collections won't be easily replaced with std collections.
- Blink's `String` wouldn't be easily replaced with `std::string`. This is because (1) Blink and V8 want to share the backing buffer of the `String` and because (2) Blink wants to use `AtomicString` for performance.
- Blink's `RefPtr` wouldn't be easily replaced with `ref_ptr` in src/base/. This is because Blink wants to use `PassRefPtr` for performance (though we could introduce `pass_ref_ptr` to src/base/).
- Blink's memory allocator needs to be fully optimized for Blink's workload, so we don't necessarily want to share allocators between Blink and Chromium. Currently Blink mixes PartitionAlloc, Oilpan, tcmalloc and system allocators, and we're planning to remove tcmalloc and system allocators (i.e., unify the allocators in Blink into PartitionAlloc and Oilpan). On the other hand, Chromium will keep using tcmalloc.

In a nutshell, I think that we should share more code between Chromium and Blink after the repository merge, and sharing src/base/ would be a good first step. However, the amount of code we really can share may not be as much as we expect.

Either way, the first priority is to complete the Blink componentization v1 as soon as possible before starting sharing more code. Sharing more code would be a topic for Blink componentization v2.

## Removing public APIs

Regarding sharing more code between Chromium and Blink, a more radical idea would be to completely eliminate public APIs and treat Blink as a part of content/.

I'm not sure if we want to do this at the moment. At the very least, this is a topic for Blink componentization v2. We can revisit this issue after completing the proposed architecture.

### How to migrate OWNERs to a new directory structure
Given that the directory structure of the proposed architecture is different from that of the current architecture, it is not clear how to migrate OWNERs to the new directory structure. For example, platform/ will be removed and merged into base/, core/web/ and modules/web/.

My answer would be we can do whatever makes sense when we actually change the directory structure, cc-ing all people in the OWNERs. Since the current OWNERs contain inactive committers, this could be a good opportunity to clean up the OWNERs :)

### Directory structures in base/
wtf/ and platform/ (except for platform/exported/) are going to be moved to base/. The simplest option for the directory structure would be just to create base/wtf/ and base/platform/, but a better option would be like this:

- base/text/ (`String` related classes)
- base/collections/ (`Vector`, `HashTable`, `Deque`, `LinkList` etc)
- base/heap/ (`Oilpan`, `PartitionAlloc` etc)
- base/utility/ (`LifecycleObserver`, `Timer`, `Task` etc)

### To what extent we want to componentize Blink
The proposed architecture componentizes Blink into three components (base, core and modules). It would be possible to split some of the components into more pieces, but what we have learned from the current architecture is that too much componentization just makes things complicated. So I don't have a plan to componentize Blink any more.

### Blink componentization v2
I'm not quite sure at the moment what the scope of the Blink componentization v2 would be (and thus I don't have a plan to start Blink componentization v2 at the moment). This will depend on how Blink is going to be used by Chromium and other things. Any ideas are welcomed.

### Anything else?
Feel free to add comments!

## Working plan
We'll start Blink componentization v1 once the repository merge is done and the core-module componentization is done. I hope it will be at the beginning of 2015 Q2.

tasak@ will use full time, bashi@ will use half time, and haraken@ will contribute to reviewing.