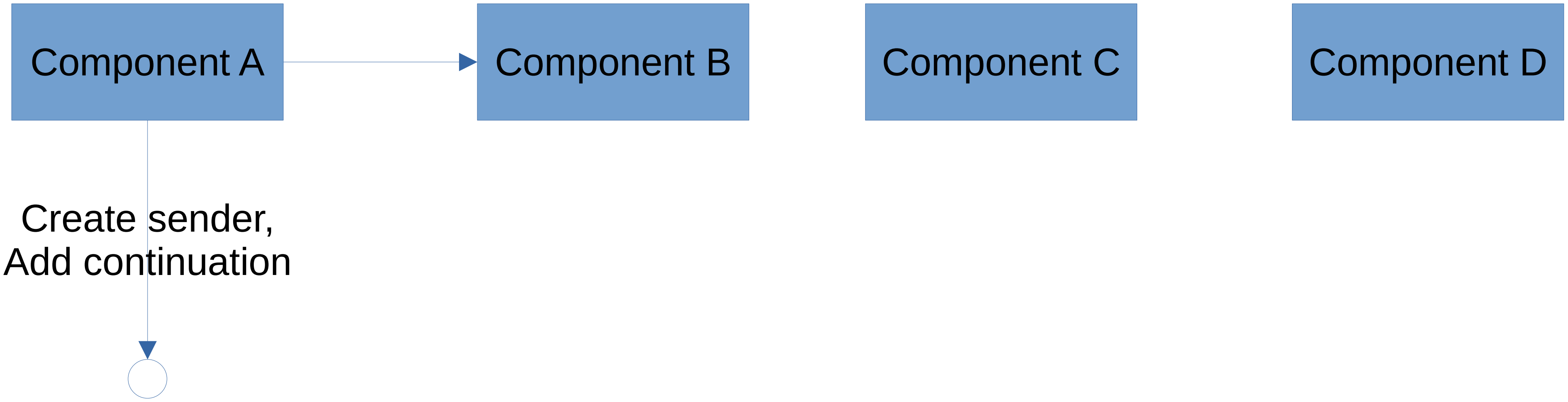
Senders and Receivers

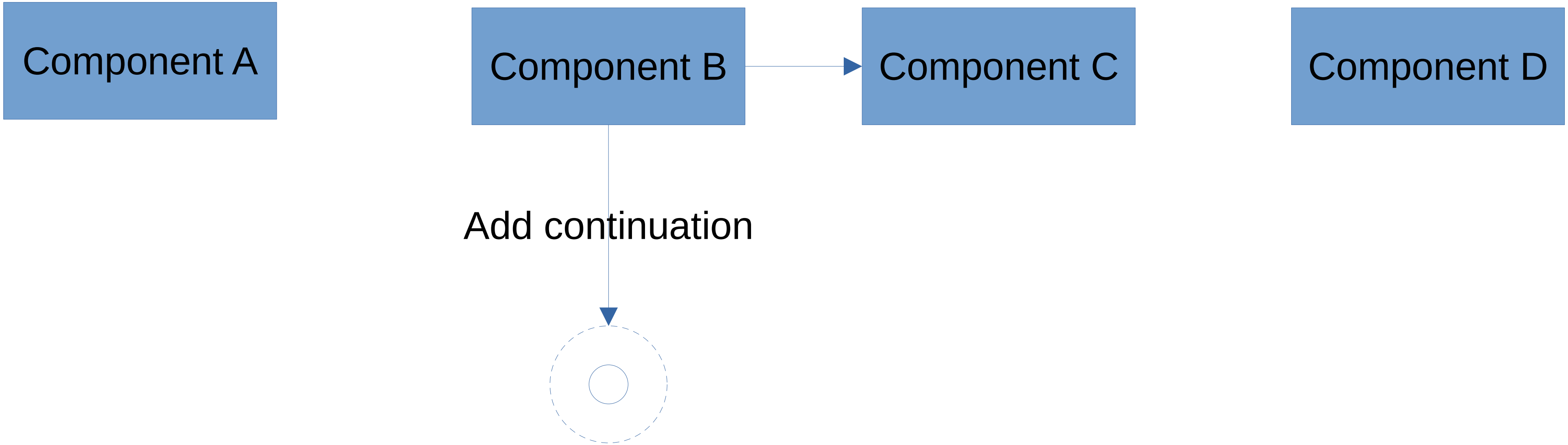
P2479

Composition, for real

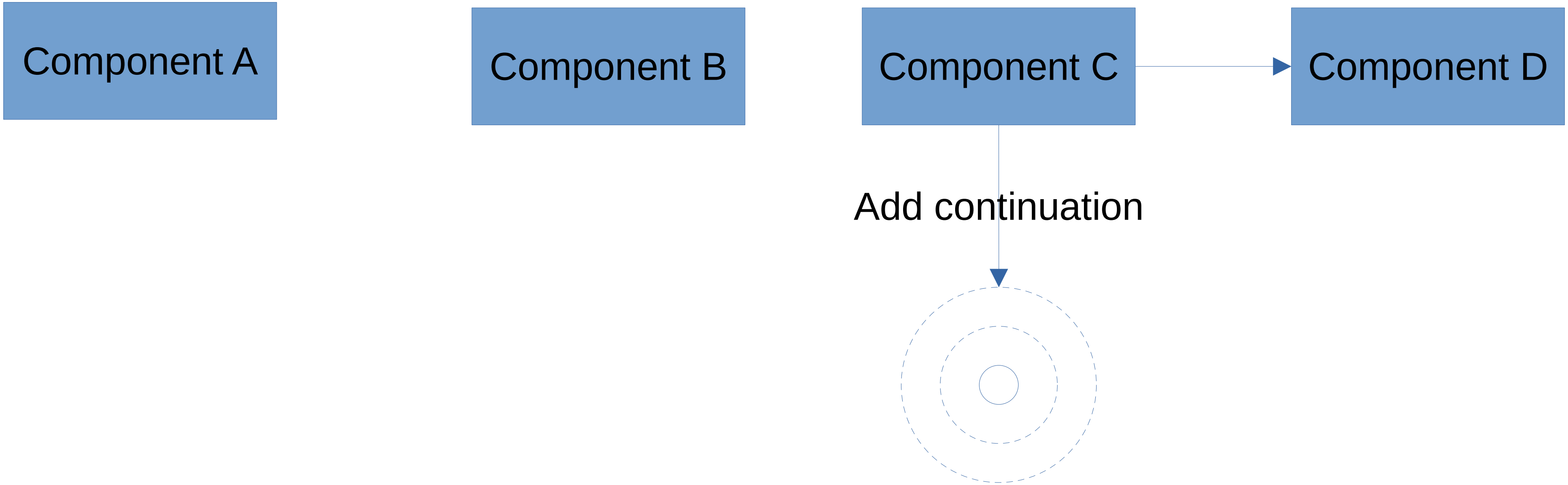
Composition across multiple layers



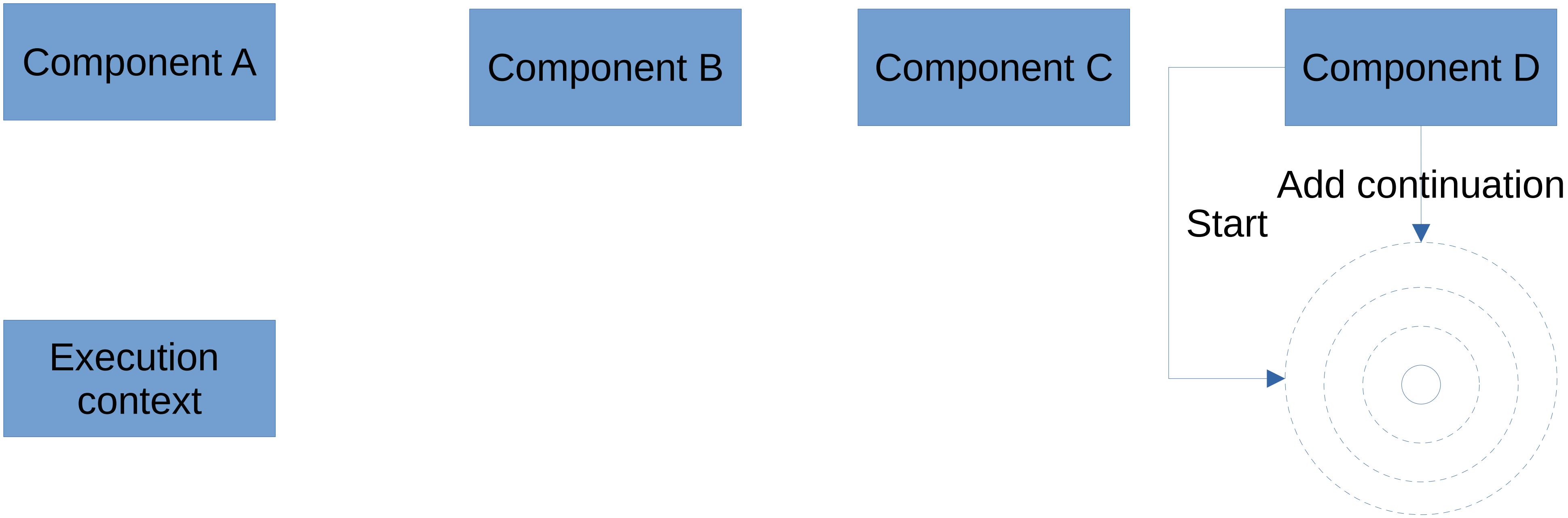
Further layers add their own continuations...



...without having to know about previous or next ones



The work graph is run once it’s complete...



Component A

Component B

Component C

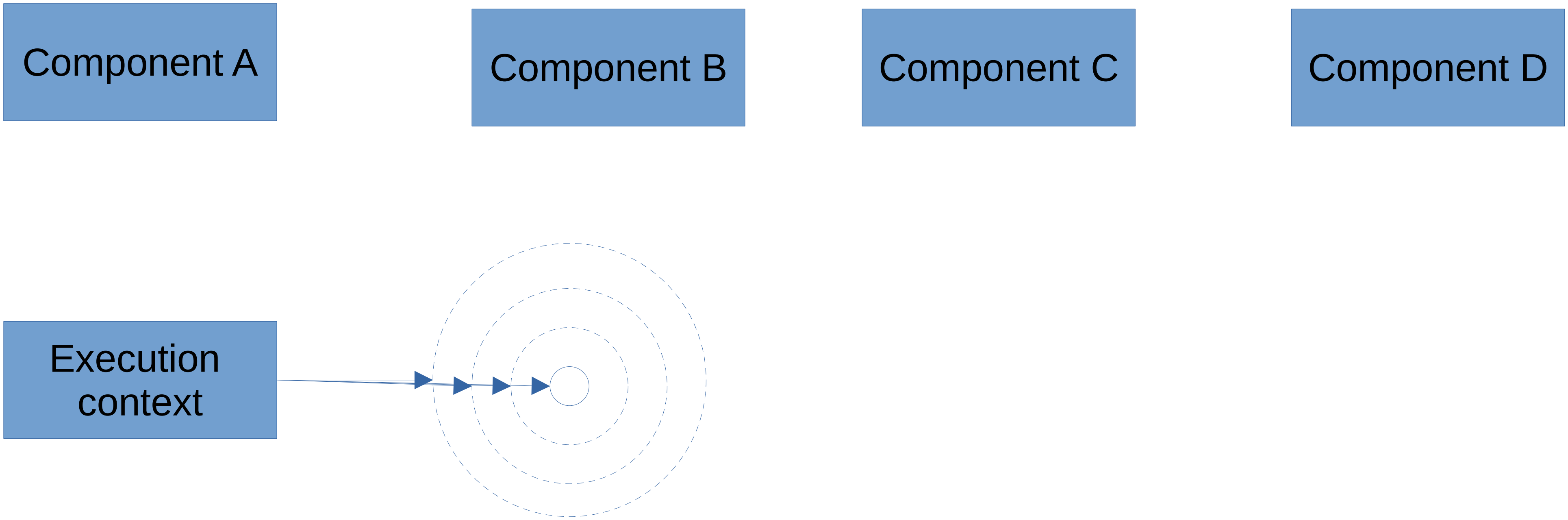
Component D

Add continuation

Start

Execution context

...on a context that none of these components created



Component A

Component B

Component C

Component D

Execution context

What does this buy us?

- Separation of concerns
  - The components don’t know about the continuations of the other components (or other algorithms applied in them)..
  - ..but separate algorithms can be applied that affect how the continuations are combined.
  - The execution context is also separate, and can be changed without affecting the rest of the code.

It’s more than just dumb wrapping

- The algorithms can deal with values and errors.
- They can intercept calls, divert calls, filter calls..
  - ..and they can filter, translate, and otherwise process the value arguments..
  - ..and error arguments.

It fits into the same framework

- The algorithms are generic; applying them in one component doesn’t change the code in another component.
- The senders and algorithms form a common vocabulary.

An executor can’t do this

- All there is for an executor is “dumb wrapping”..
- ..but that can’t deal with the values and errors.
- A refined executor maybe could, but then we have an infinite set of different ad-hoc frameworks with no common vocabulary.

P2469 doesn’t address any of this

- Yes, I know that an executor is “just the tail call completion”; to the calling client, that’s The Most Important Thing, not a hidden implementation detail.
- A completion_handler exposes an associated executor, neither of them has a common composable API that allows filtering, intercepting, chaining and translating the operations using a common API and common vocabulary.
- So, nice try, but it doesn’t resolve any of the concerns.

Let’s go for a frickin’ Pony Stable

- So, I want to make my program algorithm-pluggable, adaptable, with a common API:

NetTS	Roll your own, define asynchronous operations that have a pluggable common API.
Senders and Receivers	The common API is built-in, and used throughout.

Let me translate that for you, to plain&frank Ville-speak

- So, I want to make my program algorithm-pluggable, adaptable, with a common API:

NetTS	Invent your own API and hope that other people use the same API. This wish is unrealistic.
Senders and Receivers	The common API is built-in, and used throughout.

Let’s rephrase that once again

- So, I want to make my program algorithm-pluggable, adaptable, with a common API:

The approach	I can realistically expect to use the same algorithms and thus similar code over different work abstractions and execution context abstractions, everywhere, globally, across the entire C++ user base?
NetTS	Yes ( ) No (x)
Senders and Receivers	Yes (x) No ( )

Conclusion

- The NetTS design is so model-agnostic that it doesn’t really <u>have</u> a model, and it doesn’t establish a common API and a common vocabulary
  - but it has parts that make it not play together with our best understanding of such a common API, since it has P0443 executors in it.
- S&R does provide a common model, a common API, and a common vocabulary.

Here’s a bonus point

- Write me a piece of code that takes any asynchronous work result and posts it onto a GUI event loop.
- What do you need to write?

Here’s a bonus point

- With senders and receivers, you
  - adapt your event loop to be a scheduler
  - you take your sender that represents your work
  - and then you transfer() it.

- This works with any piece of work. Always the same. Just transfer() it. A bazillion different things that you might run as your async work, and they all transfer the same way. Every one of them.