This post puts forward a proposal for handling default account contracts in a full Account Abstraction scenario, relying on contract classes and briefly touching on upgrades as well.

Contract classes vs instances

Let's recap contract classes vs instances (as defined by Starknet) before jumping into default accounts. The idea of this separation is having contract code (ie "classes") being a first class citizen in the protocol. In this scheme, you'd first upload your contract code (identified by its hash), and then create contract instances that point to that code. Creating a contract class should be expensive, creating an instance not so much.

We could also use this separation to simplify upgradeability. Upgrading the code for a contract instance now means changing its contract class to a different one. This could be done via an opcode that changes the contract class of the current address (inspired by RSK's [CODEREPLACE

](https://github.com/rsksmart/RSKIPs/blob/master/IPs/RSKIP33.md)). Contracts that are not meant to be upgradeable would just not include that opcode.

Note that, if we also include an opcode similar to Starknet's [library_call

](https://docs.starknet.io/documentation/architecture_and_concepts/Contracts/system-calls/#library_call), we can probably remove delegate calls from the system. But that's a topic for another thread.

Default accounts

In an AA world, having a default account implementation allows anyone to interact with a user before

they have deployed their account in the system. This gives us a UX similar to Ethereum: I can create an address being offline, and share it with other people who can send funds to me before I send my first transaction.

One option to achieve this is to allow accounts with no code at all. But this means that the AA interface cannot include methods that are required for receiving

txs (eg "encrypt this private note that I'm sending you"). We could place the burden of dealing with this in the contract developers, but this would mean that whenever a contract is interacting with an address, it'd need different paths depending on whether there is code there or not (and we know from Ethereum that it's not good devex). So we'll rule out this approach, unless the resulting AA interface has no "receiving" methods.

Another option is to enshrine a default implementation at the protocol level. This way, if a contract is interacting with an address that doesn't have code yet, the protocol falls back to a default implementation. This could work, but settling on a protocol-wide implementation that can serve all default use cases could be tricky (eg what family of keys should we use for authentication?). It also brings app-level concerns down to the protocol, which it'd be best to avoid.

So we want users to be able to choose which account implementation they want to use, but without having to submit a tx to do so. We can do this by encoding the default contract class they want to use as part of their address

. This way, an Aztec address is the concatenation of a contract class identifier and an account identifier (possibly derived from a public key?).

Can we compress the contract class identifier for this purpose so addresses can still fit within a single field? Maybe account contract classes need to be registered somewhere and they get an autoincremental id as an alias? Or should we have each address be two field elements at the protocol level, and concatenate them at the UI only?

When interacting with an address, if no contract instance has ever been deployed to it, then the protocol will use the code referenced by the identifier in the address itself. Eventually, the user can upgrade this account to a different one: this would set the contract class pointer in their account, which takes precedence over the one in their address. We can assume that most account contract implementations will have a CODEREPLACE

flow to enable this.

As a reference, EIP4337 faces a similar problem when relaying user txs. If the user doesn't have an account contract yet, they need to supply an [initCode

](https://eips.ethereum.org/EIPS/eip-4337#required-entry-point-contract-functionality) as part of the tx, so there's code to interpret the user operation. This is different from our situation since we'd also need an initCode

for a user that receives

a tx, but it's similar in that it shows the need to specify the code for an account contract before it exists.

Nullifier keys and upgrades

While it’s still being defined, it’s possible that the AA interface will include methods for retrieving the nullifying key for that account. However, this key needs to be immutable to prevent double spends. This means that we’ll need to ensure that 1) the function to retrieve this key is pure (doable via introducing a sort of PURECALL

opcode), and 2) it remains the same across upgrades (doable by checking function identifiers).

Alternatively, we can keep the nullifying key out of the AA interface, and just derive it from the address at the protocol level. The derivation algorithm could even be another parameter encoded into the address itself, this time restricted to a set of a few options dictated at the protocol level.