# Activity Contexts

## Improving Modularity in Blockchain-based Smart Contracts using Context-oriented Programming

Toni Mattis
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

## ABSTRACT

Smart contracts formalize and automate interactions among and between individuals and systems in an executable and decentralized way. They are analogous to objects in object-oriented programming, but their behavior and state is replicated across multiple participants in a network and messages sent to the "object" are relayed to all network participants, allowing everyone to keep its replica up-to-date. Originally introduced in the mid-1990s, their recent surge in popularity is linked to a rising interest in blockchain-backed, general-purpose smart contract platforms.

Manging contract-specific state and behavior associated with the interacting parties and shared objects is a modularity challenge in smart contracts. Each contract has individual requirements for the (non-contract) objects it interacts with. We observed that smart contracts tend to manage object-specific state and behavior itself, often leading to a single monolithic *mediator*.

We aim at improving encapsulation and separation of concerns by allowing programmers to modularly express instance-specific state and behavior within the scope of a so called Activity Context. Activity Contexts are an extension to objects that collect these modular adaptations and jointly overlay them over instances that participate in the activity modeled by the smart contract.

We demonstrate the benefits of Activity Contexts by refactoring an exemplary smart contract and discuss their trade-offs compared to traditional object-oriented decomposition and their integration into an existing layer-based context-oriented ecosystem.

## CCS CONCEPTS

• **Software and its engineering → Object oriented languages**; *Abstraction, modeling and modularity*; Domain specific languages;

**ACM Reference Format:**
Toni Mattis and Robert Hirschfeld. 2018. Activity Contexts: Improving Modularity in Blockchain-based Smart Contracts using Context-oriented Programming. In *10th International Workshop on Context-Oriented Programming (COP'18), July 16, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242921.3242926


*COP'18, July 16, 2018, Amsterdam, Netherlands*

ACM ISBN 978-1-4503-5722-7/18/07…$15.00
https://doi.org/10.1145/3242921.3242926

## 1 INTRODUCTION

In object-oriented programming, objects manage state and behavior associated with their responsibilities and roles in a system.

If objects participate in multiple activities, they either accumulate activity-specific responsibilities, which can inflate them – or an additional object that reifies the activity itself (mediator) maintains state and activity-specific behavior for *each* participating object. In the extreme case, objects are reduced to their identity while mediators maintain all implementation details of objects and their interactions, which leads to a drastic loss of modularity with regard to information hiding or encapsulation.

In practice, programmers can often choose a trade-off and settle with a sensible mixture of both styles. However, we observed that the extreme case in which all state and behavior is concentrated in mediators, is prevalent in *smart contract development.*

*Smart Contracts.* Smart contracts were designed as a mechanism to algorithmically formalize and cryptographically secure inter-agent interactions across a network. In their modern incarnation, they are analogous to objects in object-oriented programming, but with code and state stored, executed, and verified simultaneously by many participants of a network. Messages sent to the object are replicated to all participants, allowing them to agree on the contract's state, prove to third parties that contractual obligations have been met, and automatically act on behalf of the transacting parties by, e.g., transferring money, ownership, or digital goods.

*Problem Statement.* Smart contracts run on a platform, which itself is replicated across the network. Modularity problems arise when objects provided by the platform or already existing smart contract implementations need to be extended to accommodate for new contracts deployed to that platform. For example, platform users are provided in the form of read-only *account* objects, whose code the contract developer cannot modify. As a consequence, the next plausible design choice is to handle all user-specific state and behavior in the contract itself, turning it into a large *mediator.*

*Goal.* Our goal is to provide a mechanism to express that *in the context of a specific activity/contract* an object should behave *as if it was extended by activity-specific state and behavior.* We do not eliminate the mediator – it can still manage state and behavior that does not belong to any individual object.

*Approach.* Layers in context-oriented programming provide a way to adapt a set of objects by providing partial state and behavior at run-time. This is a highly desirable feature in smart contract

development, as the contract itself models an *activity*, for which several *participating objects* need additional state and behavior.

To facilitate the tradeoff, we propose a new entity, called *Activity Context*, which has both object and layer personalities. A smart contract becomes an instance of such an activity context and can both handle messages like an object and broadcast activity-related state and behavior to participating objects, making them behave as if these new responsibilities were added to their implementation.

In addition, we argue that smart contracts tend to contain other mechanics that can be better expressed using context-oriented layers and show how the classical layer concept integrates with activity contexts. Our prototypical implementation on top of a Smalltalk-based smart contract platform is outlined.

## 2 BACKGROUND

This work relies on some background from the field of smart contracts as well as context-oriented programming. First, we describe the notion of smart contracts that is being used throughout this paper, we then give a small taxonomy of layers and their activation mechanisms which later helps to classify activity contexts.

### 2.1 Smart Contracts

Contracts as a way to document a set of mutual promises are basic building blocks of today's economy. With the widespread adoption of the World Wide Web in the 1990s, more and more resources have been managed remotely and digitally, which sparked the idea that contracts should be specified in an executable way as well - so that the contractual "algorithm" could be executed and verified automatically and act on behalf of the participating agents by transferring a digitally managed resource such as money, data, ownership rights, access permissions, licenses, etc. Szabo et al. published the first descriptions of this idea and also coined the name *smart contract* [5].

In the context of this work, we define a smart contract as a distributed object having four levels of operation:

*Logical level.* On this level, a smart contract behaves like a single *object* in object-oriented programming with state and behavior. *Network participants* (external actors and other contracts) send *messages* carrying a name and, optionally, arguments to the contract. A contract can reject or react to a message, possibly altering its state and sending messages to other contracts while doing so. Contracts and external actors have an identity, typically modeled as a an *address* bit string. There are "meta-messages" for contract creation that have no receiver, and their sender becomes a smart contract's *owner*.

*Identity and authenticity protocol.* Smart contracts require stable, unforgeable identities of participants. External actors' addresses are linked to *public keys* for which they possess a *private key*. Messages sent by them are authenticated by a cryptographic signature. Since contract creation is initiated by an external actor (either by manufacturing a contract or sending a message causing one contract to construct another one), the contract address is derived from the original sender's address and a sequence number.

*Distribution perspective.* The above behavior is realized by *replicating* contract code and state across a network of nodes. If we speak of *sending a message to a contract* from the logical perspective, this means that the message is physically sent to *all* nodes of the network to give them the chance to update the contract state.

*Consensus protocol.* To ensure that every network node receives all messages in the same order, the underlying message exchange protocol needs to address several distributed systems challenges, such as correcting for node and transmission link failures, duplication, loss, alteration of sequence or content, etc. Depending on the *threat model*, this includes defenses against intentional alteration in transit and on network nodes, and collusion of multiple nodes - formalized as the Byzantine Generals' Problem.

For the scope of this work, we will discuss smart contract examples from the logical perspective, i.e., the *contract code* to be executed. However, the logical view reifies some elements from the underlying mechanisms, which we discuss in section 3.

### 2.2 Layers

In context-oriented programming [3], a *layer* is a meta-object that, when *active*, adapts state and behavior for a set of objects simultaneously by redirecting message dispatch through so called *partial methods*. Layers are composable at run-time.

A key difference in layer concepts is their scope of *activation*, i.e., the set of messages that are dispatched to the layer rather than the object's own base methods. We identify three variation points in such a mechanism:

**Senders** The set of objects from which the layer appears active. In the most general (global) case, layer activation is independent from the sender, as illustrated in Figure 1 (a) and (b). In the most specific case, there is a single instance from which outgoing messages are dispatched to the layer – Figure 1 (c) and (d).

**Receivers** The set of objects at which the adaptation is effective. In a general case, all objects are modified, or at least all instances of a class to which the layer provides an adaptation, illustrated in Figure 1 (a) and (c). The most specific case is a single object that is adapted by a layer, as in Figure 1 (b) and (d).

**Activation Propagation** The set of objects to which layer activation can spread. A common mechanism is the *dynamic extent* (control flow), illustrated in Figure 1 (e), i.e., once the layer is active for one message dispatch, it remains active for all control flows originating from the method. A realization is the `withLayerDo: [...]` construct in *ContextS2* which activates a layer for the given block. Another mechanism is propagation via object graph, e.g., when activating the layer for a UI component, all child components are in scope as well – see Figure 1 (f).

Another difference in layer concepts is their capability to manage state [4]. *Stateful* layers can carry fields that are specific to the layer instance (all partial methods can access a field that is either layer-private or shared with other layers) or linked to adapted instances (the object itself appears to have a new field which the layer can use and expose). The mechanisms are illustrated in Figure 1.

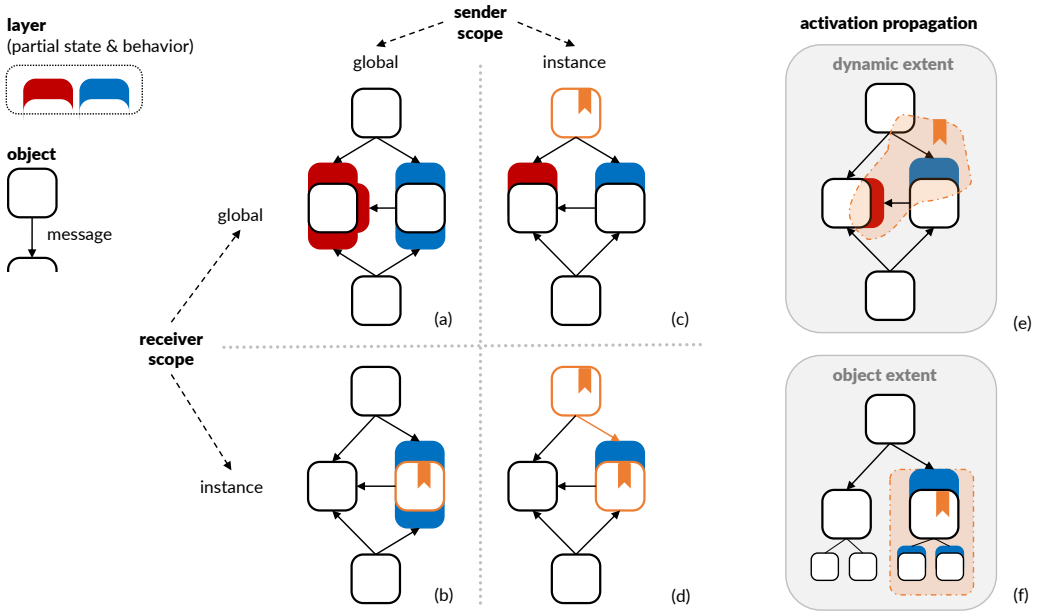


**Figure 1: Taxonomy of activation mechanisms for context-oriented layers. The quadrants in the center illustrate combinations of scoping mechanisms: (a) global activation, (b) activation around a singe receiver and independent of sender, (c) activation from the perspective of a single sender and independent of receiver, (d) activation for specific sender/receiver-pairs only. (e)/(f) mechanisms how scope can be extended once activated. Participating objects are flagged.**

## 3 EXAMPLE: VOTING WITH DELEGATION

We demonstrate modularity issues and proposed refactorings using context-oriented concepts at an exemplary smart contract that implements *voting with delegation*[1]. For brevity and prototyping reasons, we use *Smalltalk 80* syntax [1]. In practice, languages specifically designed for smart contracts, e.g., Solidity, are being used.

*Managing a poll.* Our contract named `Ballot` should keep track of voters, proposals, and votes. The contract owner is able to add proposals and voters, and can start the poll with a timeout, after which the poll is being closed and the winning proposal can be computed.

**Listing 1: Administrative methods in the voting contract**

```
Ballot >> initialize
  "run when smart contract is being deployed"
  self owner: sender;
    voters: Dictionary new;
    proposals: OrderedCollection new;
    isOpen: false.

Ballot >> addVoter: aUser
  <public>
  self assert: self owner == sender.
  self voters
    at: aUser id
    put: Voter new.

Ballot >> addProposal: aString
  <public>
  self assert: self owner == sender.
  "proposals fixed once voting started:"
  self assert: self isOpen not.
  self proposals add: (Proposal named: aString).

Ballot >> openFor: duration
  <public>
  self assert: self owner == sender.
  self assert: self isOpen not.
  self isOpen: true.
  (self after: duration "seconds") isOpen: false.

Proposal (class) >> named: aString
  "creates proposal with given name and voteCount = 0"

Voter (class) >> new
  "creates voter with weight 1 and no votes or delegates."
```

*Public methods and senders.* Since messages can be sent to a deployed smart contract from anywhere, the `<public>` pragma identifies methods that handle those external messages asynchronously. Any code path in a contract starts in public methods or initialization/destruction code, but can call non-public methods and other contracts' public methods from there.

The `sender` pseudo-variable represents the external actor or contract from which the original public message was sent. Underlying infrastructure authenticates the sender using public key cryptography and relays the message to all replicas of the contract. The sender of the initializer consequentially represents the external actor on whose behalf the contract was deployed.

[1]Inspired by the voting contract in https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html#voting, retrieved 2018-07-06

*Casting votes.* In order to cast a vote, a voter transfers its voting weight (which defaults to 1 but can be increased or nullified through delegation) to a proposal. Since anyone can send a `voteFor:` message, an eligibility check happens before, implemented by testing presence in the `voters` dictionary. Error propagation is implicit, as senders can obtain the failed assertion or error message by replaying the contract code themselves with the sequence of messages agreed on by the consensus mechanism.

Voters store the proposals they voted for. This allows to implement delegation after the delegate already voted by forwarding the transferred voting weight to the proposal's votes.

**Listing 2: Casting a vote**

```
Ballot >> voteFor: proposalId
  <public>
  self assert: self isOpen.
  self voters at: sender
    ifPresent: [:voter | | proposal |
      voter hasVoted
        ifTrue: [self error: 'already voted'].
      voter hasDelegated
        ifTrue: [self error: 'vote delegated'].
      proposal := self proposals at: proposalId.
      proposal addVotes: voter weight.
      voter proposal: proposal]
    ifAbsent: [self error: 'not eligible']
```

*Delegation.* Voters can delegate their vote to any other eligible voter. Receiving voters (delegates) increment their own weight by the transferred amount. Delegates that delegated themselves will pass the weight on, and delegates that voted already will retroactively increase the number of votes for their own proposal. Having delegated, having voted, and accumulating voting weight are mutually exclusive, and delegation loops are forbidden.

**Listing 3: Delegating a vote**

```
Voter >> credit: weight from: origin
  self assert: self ~= origin. "avoid loop"
  self hasVoted
    ifTrue: [self proposal addVotes: weight]
    ifFalse: [self hasDelegated
      ifTrue: [self delegate credit: weight from: origin]
      ifFalse: [self weight: self weight + weight]]

Ballot >> delegateTo: aUser
  <public>
  self assert: self isOpen.
  self voters at: sender
    ifPresent: [:origin |
      origin hasVoted ifTrue:
        [self error: 'already voted'].
      origin hasDelegated ifTrue:
        [self error: 'already delegated'].
      self voters at: aUser
        ifPresent: [ :target |
          origin delegate: target.
          target credit: origin weight from: origin]
      ifAbsent: [self error: 'delegate not eligible']
    ifAbsent: [self error: 'origin not eligible']
```

*Modularity in the example.* We directly observe a number of modularity issues in the above code. Besides a general tendency of the `Ballot` class to accumulate too much responsibility by acting as a mediator, the following problems will be addressed in this paper:

**Voter state** State relevant to each voter is held in a dictionary and looked up using the voter's object identity. Object-oriented design would prefer to not store object-specific state in the contract, which acts as mediator, but the object itself. Unfortunately, user objects are provided by the platform[2] and not extensible by smart contract developers. A better solution would at least look as if the object had voter state and behavior, and modularly encapsulate its implementation details.

**Permission checks** A number of administrative methods check the sender's identity upfront, which is a duplication as well as a security risk if programmers forget to copy that assertion to new administrative methods. Providing the conditional behavior as single unit of modularity that can be activated at once seems desirable. We will later address how instance-specific visibility of methods can be realized with context-oriented practices.

**Contract state** The allowed messages depend on the state the contract is in, modeled by the `isOpen` flag, and causes some verbose checks which, again, can easily be forgotten. This state can be seen as a form of context that dictates which behavioral adaptations should be active. We aim at modularly expressing state-dependent variations.

# 4 ACTIVITY CONTEXTS

An *activity context* is a first-class entity that has both *object* and *layer* personalities. It provides both own methods and state as well as partial state and behavior for objects that *participate* in the activity. The scope of activation dynamically extends to all *participating* objects.

In this work, activity contexts and layers are conceptually implemented as classes and can be instantiated multiple times at run-time, each instance having their own state.

*Participation and Scope.* For now, we define the predicate *"participating"* intensionally as being passed to the activity as method *argument* or sender. The activity context follows an *instance-specific dynamic-extent* scoping model as in Figure 1 (c) and (e), i.e., its adaptations remain active when participating objects call each other. From outside the activity context, even participating objects look unadapted, e.g., the same user object can appear as `sender` in multiple smart contracts but keeps partial state and behavior perfectly separate.

## 4.1 Partial Behavior and State

We introduce a notation[3] to identify partial methods and state on specific classes. The following code provides partial method `m` for class `C`:

```
Activity >> C >> m
  ^ result
```

For partial state, we chose a notation that directly implements the uniform access principle by only providing accessors rather than an actual local variable:

[2] In Ethereum, they are called *accounts* and are fully represented by a 256-bit address
[3] Depending on tooling, programmers might never see this notation at development time, since it can be displayed as special category of methods or separate pane in a Smalltalk browser.

```
Activity >> C >> state
  <activityAccessor>
```

Such a method would generate a getter `state` and setter `state:` at instances of class `C`.

*Activity contexts as stateful mediators.* An activity context can play the role of a *mediator* between a set of interacting objects without violating uniform access or information hiding principles. In our voting example, the voters state is managed as a dictionary and state lookup needs to interact with this implementation detail explicitly:

**Listing 4: Managing voter state**

```
"initializing state"
  self voters: Dictionary new

"check if sender is registered voter"
  self voters containsKey: sender

"check sender's voting weight"
  (self voters at: sender) weight
```

However, making the `Ballot` an *activity context* allows us to attach the relevant state directly to the `User` instances:

**Listing 5: Managing voter state within activity contexts**

```
"declaring state"
Ballot >> User >> eligible
  <activityAccessor>

Ballot >> User >> weight
  <activityAccessor>

"check if sender is registered voter"
  sender eligible

"check sender's voting weight"
  sender weight
```

This can prove useful to decouple further interactions with these objects, as all code paths originating from the activity context can see the newly added accessors and do not need to ask the mediator for this data.

## 4.2 Composing Activity Contexts and Layers

Activity contexts are composable with a range of layer types that follow the above scoping taxonomy. For example, *roles* can be expressed by a sender-specific layer that is active for all control flows originating from the senders playing the role. Receiver-specific layers can be used to model time-varying state by providing state-dependent partial methods.

*Roles.* In our example smart contract, we encountered the role of the *owner* who can manage voters and proposals. We will define a layer `Owner` which defines administrative behavior as partial methods on the Ballot activity context itself:

**Listing 6: Sender-scoped Layer Activation for Roles**

```
Owner >> Ballot >> addProposal: aString
  <public>
  self assert: self isOpen not.
  self proposals add: (Proposal named: aString).

Ballot >> initialize
  Owner activateFrom: sender
  "... remaining initialization ..."
```

The `activateFrom:` message understood by all layers allows the platform to dynamically activate this layer whenever a message from that object is sent. The above code makes use of the invariant that whenever a public contract method is being executed, the layers attached to `sender` must be active.

*States.* Our smart contract is only open for voting for a specific time span, a fact which can be modeled as a receiver-specific layer that implements partial behavior that only can happen while the poll is open:

**Listing 7: Receiver-scoped Layer Activation for States**

```
Open >> Ballot >> voteFor: aProposal
  <public>
  "... voting logic ..."

Open >> Ballot >> isOpen
  ^ true

Owner >> Ballot >> openFor: duration
  <public>
  Open activateAround: self.
  (Open after: duration) deactivateAround: self.
```

This code re-uses the owner role to control visibility of the `openFor:` method while activating (an instance of) the `Open` layer for exactly this ballot. The difference between the messages `activateAround:` and `activateFrom:` is the point in time when the layer is activated: A layer attached via `activateFrom:` becomes active for all outgoing message sends of that object (instance-specific sender scope as in Figure 1 (c)), but partial state and behavior would not be visible to outside objects. In contrast, `activateAround:` would dispatch all incoming message sends to the layer and thereby make adaptations visible to outside objects (instance-specific receiver scope as in Figure 1 (b)).

*Managing eligibility.* The `voteFor:` method is constrained by two factors: First, the user needs to be eligible, i.e., registered by the owner. Second, the contract must be open for voting. We propose to model eligibility as boolean variable for each user. We make use of a default value, in this case implemented as return value of the accessor-generating method:

**Listing 8: Managing Eligibility: Boolean Variable**

```
Open >> Ballot >> voteFor: proposalId
  <public>
  self assert: sender eligible.
  "... voting logic ..."

Ballot >> User >> eligible
  <activityAccessor>
  ^false "default value"

Owner >> Ballot >> addVoter: aUser
  <public>
  aUser eligible: true.
```

Alternatively, it is possible to model eligibility as voter role:

**Listing 9: Managing Eligibility: Voter Role**

```
Open >> Ballot >> voteFor: proposalId
  <public>
  sender voteFor: (self proposals at: proposalId).

Ballot >> User >> voteFor: aProposal
```

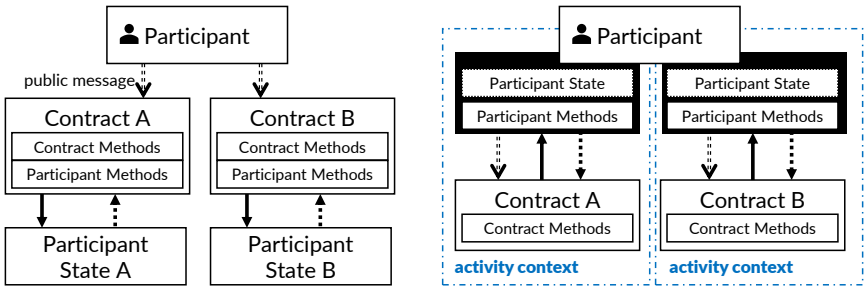


**Figure 2: Working principle of activity contexts. On the left, two mediator-style contracts own participant-specific methods and manage participant state themselves. On the right, activity contexts provide the chance to move participant-specific methods to the participating object and hide implementation details of participant state from the contract.**

```
  self error: 'not eligible to vote'.

Voter >> User >> voteFor: aProposal
  "actual voting logic"

Owner >> Ballot >> addVoter: aUser
  <public>
  Voter activateFrom: aUser.
```

This is more general and allows modular implementation of further partial behavior in the intersection of the two layers without copying an eligibility check around several methods.

The drawbacks of this approach are, besides added complexity of another layer, the fact that activation prioritization needs to rank sender-specific layers higher than the activity context, i.e., the actual voting logic overrides the error-throwing partial method of the activity context and not vice versa. From the code alone, this behavior is not explicitly visible and we defer prioritization of mixed activation scopes to future work.

**Listing 10: Casting a vote using the activity context**

```
Open >> Ballot >> voteFor: proposalId
  <public>
  self assert: sender eligible.
  sender hasVoted
    ifTrue: [self error: 'already voted'].
    proposal := self proposals at: proposalId.
    sender voteFor: proposal]

Ballot >> User >> voteFor: aProposal
  aProposal addVotes: self weight.
  self hasVoted: true.
  self votedFor: aProposal.
```

## 4.3 Explicit Activation

Since activity contexts have layer personality, they offer constructs to deal with them as layers. External code can dynamically activate an activity context to view an object from the activity's perspective. For example, another smart contract or client code could determine if the user has voted on a given issue:

```
myBallot withLayerDo: [voted := myUser hasVoted]
```

Also, activity contexts can delegate partial methods to base methods (or partial methods of other layers that are on the activation stack) using the `proceed` construct (also known as `next` in literature). Since our example was focused on extending objects and controlling message visibility, there was no situation in which proceeding to the next layer would have helped.

## 4.4 Discussion

Activity contexts, while behaving like an object, provide partial state and behavior to objects it interacts with (summarized in Figure 2). We elaborated on the following language constructs associated with activity contexts:

- partial method definitions (`Activity >> C >> m`)
- partial state definitions (via `<activityAccessor>` pragma)
- implicit activation for the dynamic extent of an interaction with participating objects
- explicit dynamic activation (via `withLayerDo:` message)

We provided a taxonomy of activation scopes and classified activity contexts as sender-scoped to a single instance – themselves, acting as their object personality.

We demonstrated that these concepts can modularly attach behavior and state to objects that are not under our control and can be of use in the programming and execution model for smart contracts.

In a smart contract setting, objects are often re-purposed by new contracts. A common design choice is to make them as minimal as possible (in practice, often just an identity, address, or cryptocurrency balance) and manage their state inside the smart contract, which acts as a mediator. Using activity contexts and other context-oriented concepts, we can restore encapsulation and move responsibilities to the objects they belong to.

*A note on notation and tooling.* Depending on how the programming environment presents the code, partial behavior can be *inlined* or attached to the respective classes and yield a composed view on shared objects from the perspective of the smart contract that is being developed.

*Interaction with normal layers.* We also showed how context-oriented layers with different activation scopes can be used to adapt the activity context itself to implement

- role-specific permissions using a sender-scoped activation (`activateFrom:`) tied to the objects that play the role and
- state-specific partial behavior activated for the smart contract as a receiver (`activateAround:`)

but left prioritization details for future work and only hinted that "role layers" and "state layers" should override the more general behavior defined in the activity context.

## 5 IMPLEMENTATION

Smart contract platforms would provide a secured virtual execution environment to run contracts in. For prototyping, however, we use an unsecured Squeak/Smalltalk execution environment to run contracts and implement activity contexts.

In this section, we will elaborate on the mechanism allowing activity-specific message dispatch, which can be used in scenarios different from smart contracts as well. The full prototype runs the contract on top of an experimental blockchain-based message replication and consensus layer, which is not discussed here.

### 5.1 Method Wrapping

In Squeak/Smalltalk, methods are meta-objects represented by `CompiledMethod` instances[4] containing the opcodes interpreted by the execution environment. Each class has a method dictionary mapping a message selector to such an executable meta-object. We use the fact that `CompiledMethod` objects can be replaced by any object that responds to the
`run: selector with: arguments in: receiver` message to implement custom method dispatch semantics.

*Adaptation providers.* Adaptation providers are *internal meta-objects* used to implement activity contexts. They store and provide partial method definitions. The method definitions that read like `Activity >> Class >> selector` in the code examples compile to a partial method that is placed in a dictionary that maps pairs of class name and selector to `CompiledMethod` objects.

Adaptation providers need two methods, one that checks whether a specific adaptation is present, and one that dispatches to this implementation. The latter invokes the compiled partial method by sending
`withArgs: args executeMethod: compiledMethod` to the object under adaptation, which itself is a primitive of Smalltalk's execution environment.

*Activation stack.* A central structure is the thread-local *activation stack* that determines which adaptation providers are active. Entering an activity context corresponds to pushing it on that stack, leaving the dynamic scope of the activity would pop it again.

*Advisable methods.* Any message selector that can *possibly* become subject to adaptation or cause activation of activity contexts or layers is associated with an `AdvisableMethod` object implementing said `run:with:in:` method.

When such a method is run instead of a `CompiledMethod`, it modifies method dispatch in the following way:

(1) Search the *activation stack* and the set of *receiver-scoped* adaptation providers. If one responds to the current class/selector combination, dispatch to it and return the result.
(2) Push *sender-scoped* and *receiver-scoped* adaptation providers onto the activation stack, they need to be active for all outgoing calls.
(3) Call the default `CompiledMethod` or, if none, raise an exception.
(4) Pop sender and receiver scoped adaptation providers and return the default method's result.

### 5.2 Scoped Activation

Objects have two new (lazily initialized) fields for receiver-scoped and sender-scoped adaptation providers. The receiver-scoping methods `activateAround:` and `deactivateAround:` add to and remove from the set of *receiver-scoped* adaptation providers. From the standpoint of our taxonomy from Figure 1, all instances that have an activity context or layer in this set, belong to the receiver scope of that layer and are affected by its modifications independently of which object was the message sender.

In contrast, `activateFrom:` and `deactivateFrom:` add to and remove from the set of *sender-scoped* adaptation providers. In our taxonomy, all instances that have certain layer in this set, belong to the sender scope of that layer and can potentially affect any message receivers down the call graph.

### 5.3 Compiler Modifications and State Handling

*Compilation side-effects.* To provide a partial method for a method that does not yet exist at the target class, each compilation of such a method would transparently put an empty `AdvisableMethod` object at the corresponding slot of the target class. This ensures that the runtime always checks for partial methods.

*Pseudo-variables.* We also rewrite a number of expressions. For example, the `thisActivity` pseudo-variable is replaced by a message send to the current thread
`Processor activeProcess thisActivity`, which then responds with the value of a thread-local variable that was previously set during dispatch to cache the adaptation provider for the active partial method.

*Accessors.* The code generated by an `<activityAccessor>` pragma, e.g.,

```
Activity >> C >> state
  <activityAccessor>
  ^ 42
```

would closely resemble the following getters and setters that manage the state as dictionary of instances and their instance variables. The dictionary is attached to the enclosing activity, which can be retrieved using the `thisActivity` pseudo-variable.

```
Activity >> C >> state
(thisActivity instVars at: self) at: #state ifAbsent: 42.

Activity >> C >> state: anObject
(thisActivity instVars at: self
ifAbsentPut: [Dictionary new])
at: 'state' put: anObject.
```

Note that `self` refers to an instance of `C`, while `thisActivity` refers to an instance of `Activity`.

[4] We use the term *meta-object* to refer to objects that *constitute* a program, to distinguish them from the *domain* objects created and handled by the program.

*Modifying read-only objects.* At this point, we would like to emphasize that *we do not modify* the `c` instance. We modify a contract-owned dictionary `instVars` that stores the state of said instance like the mediators did before, but access to that state is provided by `c state` and `c state: value` given `c` is an instance of `C`.

As a consequence, state access appears *uniform* to the contract code, regardless of the storage location. Moreover, any other object interacting with `c` in the same context does not need to ask `Activity` for `c`'s state but can directly call `c state`.

However, due to restrictions in our programming language we cannot prevent the contract from accessing `instVars` and bypassing encapsulation in this prototype.

In conclusion, this implementation strategy covers a wider range of possible scenarios than required by activity contexts as it allows to specify adaptations for effectively any pair of objects exchanging messages, but allows us to quickly integrate activity contexts and layers with the existing smart contract platform. We do not support layer composition via `proceed` yet and need to extend tooling to better present the new units of modularity to programmers.

## 6 FUTURE WORK

We identified a few limitations that can be addressed in future work on activity contexts. To us, the most interesting ones include:

*Priorities.* Prioritization of differently scoped activations is currently unresolved and implementation-specific. There is a conflict between sender-scoped and receiver-scoped layers: If the activation stack reflects the call stack, as in our simple implementation, sender-scoped partial definitions could be overridden by purely receiver-scoped definitions if the receiver's layers have been activated via side effect from another control flow. In our case of smart contracts, the opposite can also be desired: A sender having an administrative role attached via layer might want to override default methods provided by a layer around the receiving objects. How to control such conflicts remains an open question.

*Explicit participation.* We currently imply that an object participates in an activity when it has been passed as argument or sent the message to the activity context itself. While this is desirable behavior for smart contracts, one could also extend the concept to support explicit participation, e.g., only when the object passes through a specific part of the interface, or after it is explicitly "wrapped", it is viewed as participant, and as unadapted object otherwise.

*Explore composability in smart contracts.* We see several potential use-cases for re-use of smart contract code. For example, the delegation mechanism could have been implemented inside a layer and activated at run-time on behalf of the owner of that smart contract. This way, smart contract mechanisms and patterns can be provided as layers and composed like traits, except that they can provide cross-cutting partial behavior and can be activated at a more fine-grained level.

## 7 RELATED WORK

There are paradigms based on object-oriented programming that provide similar mechanisms as activity contexts. None of them have been explored in the smart contract domain yet:

*Subject-oriented programming (SOP).* SOP [2] can provide different sender-specific views on objects. In a sense, activity contexts as well as sender-scoped layers can be seen as a general form of subject-oriented programming.

*Role-oriented programming (ROP) and Data, Context, Interaction (DCI).* The ROP paradigm [7] and DCI pattern [6] describe activity-specific state and behavior modularly and also solve the problem that activities either cross-cut objects and make their code less coherent, or facilitate large mediators. Especially in DCI, objects only carry essential state, which aligns well with the design of smart contract platforms. It appears as if activity contexts, DCI, and ROP can, to some extent, emulate each other for the use cases described in this work. Activity contexts do not model roles explicitly, but rely on the more general notion of a layer to implement roles and states.

## 8 CONCLUSION

Smart contracts have a range of modularity challenges that are unique to the fact that decentrally developed contracts share and repurpose objects without having the chance to adapt them, their logic often deals with roles and states, and their decentralized execution model makes message senders as important as receivers.

We are confident that context-oriented programming concepts, such as layers, can plausibly address these modularity challenges. We presented activity contexts as a new concept derived from layers that provides a good fit to model smart contracts themselves, and showed how smart contract code can be modularized using a mixture of activity contexts and layers.

## REFERENCES

[1] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[2] William Harrison and Harold Ossher. 1993. Subject-Oriented Programming: A Critique of Pure Objects. *ACM SIGPLAN Notices* 28, 10 (Oct. 1993), 411–428. https://doi.org/10.1145/167962.165932
[3] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich* 7, 3 (2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4
[4] R. Hirschfeld, H. Masuhara, A. Igarashi, and T. Felgentreff. 2015. Visibility of Context-Oriented Behavior and State in L. *ResearchGate* 32, 3 (Aug. 2015), 149–158. https://doi.org/10.11185/imt.11.11
[5] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (Sept. 1997). https://doi.org/10.5210/fm.v2i9.548
[6] Hector Adrian Valdecantos, Katy Tarrit, Mehdi Mirakhorli, and James O. Coplien. 2017. An Empirical Study on Code Comprehension: Data Context Interaction Compared to Classical Object Oriented. IEEE, 275–285. https://doi.org/10.1109/ICPC.2017.23
[7] Michael VanHilst. 1997. *Role Oriented Programming for Software Evolution.* Thesis.