Etherisc GIF Manual Documentation

Release latest

Aug 28, 2020

Contents

This manual explains the key ideas, terms, and principles of the Generic Insurance Framework (GIF) as a tool that enables to utilize both smart contracts and microservices to create specific insurance products. You will learn the best ways to interact with this framework, as well as employ the available functionality to the fullest.

The manual focuses on the needs of product builders and helps them to implement the existing solutions on top of the Generic Insurance Framework.

CHAPTER 1

User Manual for the Generic Insurance Framework

1.1 Terminology

Below, you will find a glossary of the technical terms used in this document.

Actor. Any participant of the DIP that uses the ecosystem to perform an activity on it (e.g., a product, a product owner, an oracle owner, the instance operator, etc.).

Application. Data applied by a customer requesting an insurance policy. An application is the predecessor of a policy. Not to be confused with software application; in our context we avoid the term "application" in this sense.

Claim. Data related to an insurance claim, which requires approval.

Decentralized Insurance Platform (DIP). An ecosystem supported by the DIP Foundation that unites product builders, risk pool keepers, resellers, oracle providers, claim adjusters, relayers, and underwriters.

Decentralized Insurance Protocol (DIP Protocol). A set of standards, rules, templates and definitions which define the interaction of participants in the ecosystem.

Generic Insurance Framework (GIF). A combined codebase, which includes smart contracts and utility services (core smart contracts and microservices) provided by the DIP Foundation and partners. The codebase can be extended by product-specific smart contracts and microservices created by product builders. Using this framework, product builders can develop full-featured DApps.

GIF instance. A deployed set of core smart contracts, operated by an instance operator, in most cases together with an appropriate set of utility services. A GIF instance is essentially an "Insurance as a Service (IaaS)".

Instance operator. An Ethereum account which operates an instance of the GIF. An instance operator can be a decentralized organization (DAO) or a single account owned by some legal entity.

Metadata. A shared object between all the objects of a particular policy flow.

Oracle. A service used to provide information to smart contracts from external resources, confirm certain events, and deliver particular data to a product.

Oracle owner. An Ethereum account registered on the DIP with a set of permissions for creating oracles and oracle types and performing operations on them.

Oracle type. A type of request to an oracle containing attributes that describe a request and respond to it. Oracles join an oracle type.

Payout. Data related to the expected and actual payout for a claim.

Policy. Technical representation of the legal agreement between a policy buyer and a carrier.

Policy flow. A core smart contract that represents a workflow of insurance policy life cycle, involving such steps as application underwriting, risk assessment, claim review, and payouts.

Policy token. A ERC1521 token (extension of a ERC 721 NFT Token), which represents a policy as a set of particular fields.

Product. A registered smart contract with permissions to create and manage policy flows.

Product owner. An Ethereum account registered on the GIF with a set of permissions allowing to create and manage product contracts and oracle types.

CHAPTER 2

Generic Insurance Framework

The Generic Insurance Framework represents a combined codebase for the Decentralized Insurance Platform, a basic implementation that enables users to develop blockchain-based applications.

The basic idea behind the GIF is to abstract the generic parts shared across multiple different products and leave only product-specific parts, such as risk model, pricing, and payout configurations, to be adjusted. The goal is to enable quick and easy deployment of working products.

In its core, the GIF accumulates a number of componets:

- core smart contracts
- core microservices
- product-specific smart contracts
- product-specific microservices

Essentially, the GIF has two major layers — a smart contracts one and a utility one — with DIP Foundation and partners being able to contribute to both.

The **smart contracts layer** is designed in the way that any blockchain product built on top of the GIF can be easily implemented into any network supporting the Ethereum Virtual Machine. Any product owner is able to create a full-featured decentralized app by adding a couple of simple domain-specific contracts to a number of generic ones that the framework provides.

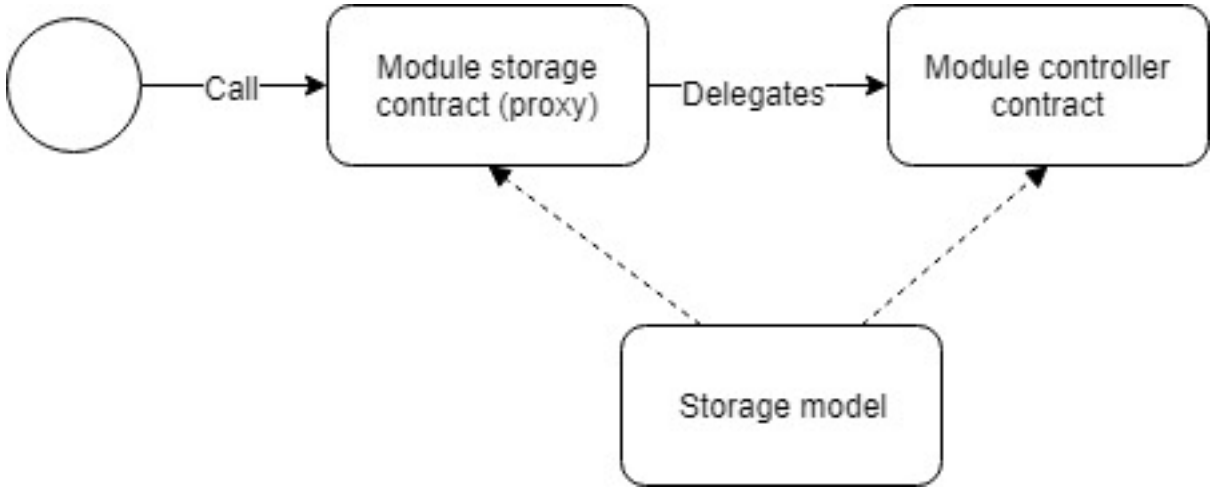
The core contracts are deployed on-chain and operate by an instance operator as a shared service for many different products. The instance operator can be a decentralized organization (DAO) or a more traditional legal entity. A product, working on top of the GIF, is a smart contract (or set of smart contracts) connected to the framework's core contracts through a unique entry point.

The DIP declares the underlying principles and requirements based on which the architecture of smart contracts is developed:

- Generic Insurance Framework provides a unified interface, which connects a product to data and decision providers (oracles).
- A product contract utilizes a simple and clear interface for integration with the GIF.

- Once the agreement is signed and a policy token is issued, parties cannot change the expected policy flow behavior. A policy life cycle should operate on the contracts, which this policy was issued by.
- Core contracts can be upgraded. This is needed to make bug fixes and add new features.

The smallest building blocks are called "modules." A "module" is a pair of a "storage" contract and a "controller." They share the same storage model and interface objects. The "storage" contract is a proxy, which delegates calls from a "storage" to a "controller," which implements basic logic (i.e., a method to change a state). This mechanism ensures that the module can be upgraded.



A "service" contract contains business logic details and defines rules (i.e., "Underwritten" is the next state after "Applied"). The "service" contract manages modules by calling controllers from storages. It is also an entry point for actors (products, oracles, product owners, etc.).

"Controllers" serve as entry points for "services." It is important to differentiate "services" behavior from that of "controllers."

The **utility layer** can contain any number of off-chain utility services supplementing the on-chain functionality. For example, statistical monitoring of events triggered by contracts, making e-mail or instant messenger notifications, accepting fiat payments for policies, as well as making fiat payouts. As a result, any product app can be fully functional on chain even without any support from the utility layer, as well as can provide a full spectrum of the required features.

The key feature to have in a framework is the ability to upgrade and replace individual elements of the system. For this purpose, the Generic Insurance Framework employs a microservices-based architecture approach for its utility layer. The GIF organizes off-chain operations as a collection of loosely coupled services, each implementing a single independent function — a state known as "decomposition by business capabilities."

CHAPTER 3

Core Smart Contracts

Core smart contracts represent a number of key contracts and modules. The product service, policy flow, and modules are described below. Core smart contracts are deployed and operated by an instance operator - a DAO or some other (legal) entity. The instance operator publishes the entry points to its instance of the GIF (e.g. the address of the Product Service) and registers actors in the instance.

3.1 Product Service

The **product service** is an entry point for a product contract. During smart contract deployment, the address of the product service should be passed as one of the constructor arguments.

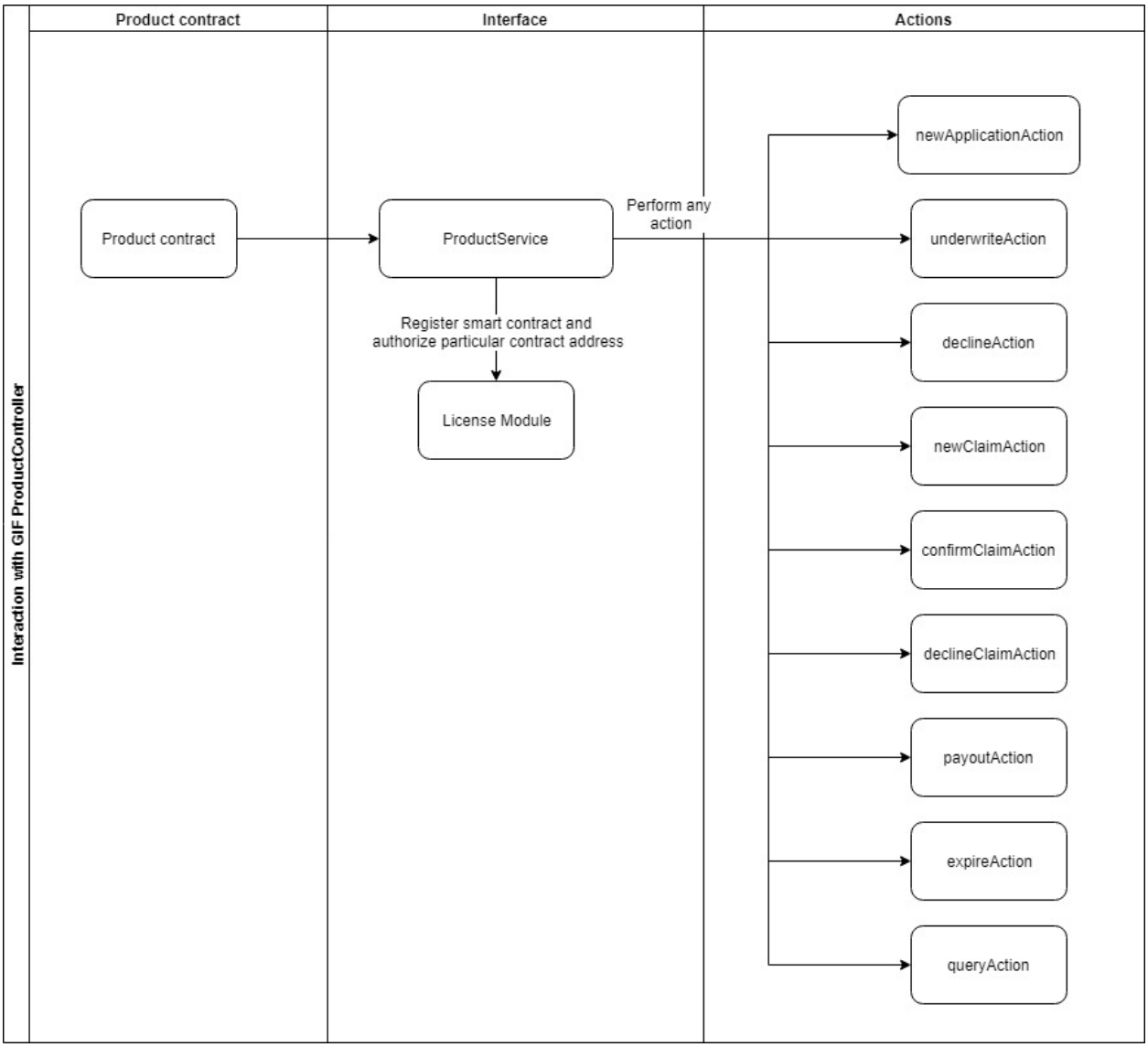
All product service methods are used by a product contract.

Below, you will find a list of the methods invoked by the product service:

- **register** is used to register new product contracts by providing a product name and specifying a policy flow. On approval, a product contract obtains access to call entry methods.
- **newApplication** is employed to store new application data, which contains such fields as premium amount, currency, payout options, risk definition, etc. A policy buyer signs a policy agreement using this method.
- **underwrite** is used to sign a policy agreement by an insurance company.
- **decline** declines an application.
- **newClaim** declares a new claim.
- **confirmClaim** confirms a claim. A new payout object is created after this.
- **declineClaim** declines a claim.
- **payout** declares payout that was handled off-chain or on-chain based on the policy currency.
- **expire** sets a policy expiration.
- **request** is used to communicate with oracles when a smart contract requires data or a decision by a particular actor.

- **getPayoutOptions** checks payout options data.
- **getPremium** checks a premium per application.

On the diagram below, you can see the actions a product service performs.



The code below illustrates how the above-mentioned methods can be invoked.

```
interface IProductService {
        function register(bytes32 _productName, bytes32 _policyFlow)
            external
            returns (uint256 _registrationId);

        function newApplication(
            bytes32 _customerExternalId,
            uint256 _premium,
            bytes32 _currency,
            uint256[] calldata _payoutOptions
        ) external returns (uint256 _applicationId);
```

(continues on next page)

(continued from previous page)

```
12
13          function underwrite(uint256 applicationId)
14              external
15              returns (uint256 _policyId);
16
17          function decline(uint256 _applicationId) external;
18
19          function newClaim(uint256 _policyId) external returns (uint256 _claimId);
20
21          function confirmClaim(uint256 _claimId, uint256 _sum)
22              external
23              returns (uint256 _payoutId);
24
25          function declineClaim(uint256 _claimId) external;
26
27          function expire(uint256 _policyId) external;
28
29          function payout(uint256 _payoutId, uint256 _sum)
30              external
31              returns (uint256 _remainder);
32
33          function getPayoutOptions(uint256 _applicationId)
34              external
35              returns (uint256[] memory _payoutOptions);
36
37          function getPremium(uint256 _applicationId)
38              external
39              returns (uint256 _premium);
40
41          function request(
42              bytes calldata _input,
43              string calldata _callbackMethodName,
44              address _callabackContractAddress,
45              bytes32 _oracleTypeName,
46              uint256 _responsibleOracleId
47              ) external returns (uint256 _requestId);
48      }
```

3.2 Policy Flow

The **policy flow contract** implements business logic for a policy life cycle. A product contract should specify a desired policy flow contract during registration. The policy flow contract has permissions to manage modules.

A policy life cycle could be defined as a "state machine." By this definition, a policy flow contract specifies transition rules between states of core objects (applications, policies, claims, and payouts) and a sequence of actions that manage the "state machine."

A policy flow contract contains the logic of how to handle the GIF contract modules and operate application, policy, claim and payout entities.

CHAPTER 4

Modules

A module represents a group of smart contracts, with each module containing at least one storage and one controller contract.

A storage contract acts as a database for the core objects. A controller contract includes an implementation that helps to manage core objects in a storage contract. In its turn, a storage contract delegates methods and makes calls to a controller contract, which modifies the state of a storage contract.

Here is the list of the modules behind the Generic Insurance Framework:

- a **policy module** (manages applications, policies, claims, payouts, and metadata objects)
- a **registry module** (registers sets of the core contracts used in a policy flow lifecycle in release groups)
- a **license module** (manages products)
- a **query module** (manages queries made to oracles and delivers responses from them).

CHAPTER 5

The license module

The **license module** stores registration data and the data related to the registered products. The module is responsible for authorization of a particular contract address and rejects calls from unauthorized senders.

The approval or disapproval of calls is managed by the responsible methods invoked by the instance operator. A product contract can be managed by any Ethereum account, be it a single account, a multisig, or a DAO.

Product contracts are registered in a smart contract, and its registration proposal is on review for the instance operator, which can then perform certain actions related to the registration of product contracts.

All **license controller methods** are used by the instance operator, except for the register method, which can be called by product owners only.

The methods invoked by the license controller include:

- **register** is used to register a proposal by a product contract.
- **declineRegistration** is called by the instance operator to decline registration.
- **approveRegistration** is called by the instance operator to approve registration.
- **disapproveProduct** is called when the instance operator wants to decline the registration, which was previously approved by it.
- **reapproveProduct** is used to approve the registration after it was declined by the instance operator.
- **pauseProduct** is employed by the instance operator to pause a product contract.
- **unpauseProduct** is used by the instance operator to unpause a product contract.
- **isApprovedProduct** is used by the instance operator to check if a product contract is approved.
- **isPausedProduct** is used by the instance operator to check if a product contract is paused.
- **isValidCall** is used by the instance operator to check if a product contract call is valid.
- **authorize** is used by the instance operator to check if a product contract address is authorized and what policy flow it uses.
- **getProductId** is used by the instance operator to check a product contract ID.

Below, you can see how to invoke all the above-mentioned methods available through the license controller.

```
interface ILicenseController {

        function register(bytes32 _name, address _addr, bytes32 _policyFlow)
            external
            returns (uint256 _registrationId);

        function declineRegistration(uint256 _registrationId) external;

        function approveRegistration(uint256 _registrationId)
            external
            returns (uint256 _productId);

        function disapproveProduct(uint256 _productId) external;

        function reapproveProduct(uint256 _productId) external;

        function pauseProduct(uint256 _productId) external;

        function unpauseProduct(uint256 _productId) external;

        function isApprovedProduct(address _addr)
            external
            view
            returns (bool _approved);

        function isPausedProduct(address _addr)
            external
            view
            returns (bool _paused);

        function isValidCall(address _addr) external view returns (bool _valid);

        function authorize(address _sender)
            external
            view
            returns (bool _authorized, address _policyFlow);

        function getProductId(address _addr)
            external
            view
            returns (uint256 _productId);
    }
```

CHAPTER 6

The policy module

The **policy module** is responsible for managing applications, policies, claims, payouts, and metadata objects. The policy module is managed by a policy flow contract.

The methods invoked by the policy controller are as follows:

- **createPolicyFlow** is called to create a new policy flow.
- **setPolicyFlowState** is employed to set a policy flow state.
- **createApplication** is used to create a new application.
- **setApplicationState** sets an application state.
- **getApplicationData** helps to view application data per application ID.
- **getPayoutOptions** is called to view payout options per application ID.
- **getPremium** is invoked to view a premium amount per application ID.
- **createPolicy** creates a new policy.
- **setPolicyState** automatically sets a policy state.
- **createClaim** creates a new claim.
- **setClaimState** automatically sets a claim state.
- **createPayout** creates a new payout.
- **payOut** is called to get data on a payout remainder.
- **setPayoutState** automatically sets a payout state.

The code below illustrates how to invoke the above-mentioned methods of the policy module.

```
1  interface IPolicyController {
2
3  function createPolicyFlow(uint256 _productId)
4      external
5      returns (uint256 _metadataId);
```

(continues on next page)

(continued from previous page)

```
6
7   function setPolicyFlowState(
8       uint256 _productId,
9       uint256 _metadataId,
10      IPolicy.PolicyFlowState _state
11  ) external;
12
13  function createApplication(
14      uint256 _productId,
15      uint256 _metadataId,
16      bytes32 _customerExternalId,
17      uint256 _premium,
18      bytes32 _currency,
19      uint256[] calldata _payoutOptions
20  ) external returns (uint256 _applicationId);
21
22  function setApplicationState(
23      uint256 _productId,
24      uint256 _applicationId,
25      IPolicy.ApplicationState _state
26  ) external;
27
28  function createPolicy(uint256 _productId, uint256 _metadataId)
29      external
30      returns (uint256 _policyId);
31
32  function setPolicyState(
33      uint256 _productId,
34      uint256 _policyId,
35      IPolicy.PolicyState _state
36  ) external;
37
38  function createClaim(uint256 _productId, uint256 _policyId, bytes32 _data)
39      external
40      returns (uint256 _claimId);
41
42  function setClaimState(
43      uint256 _productId,
44      uint256 _claimId,
45      IPolicy.ClaimState _state
46  ) external;
47
48  function createPayout(uint256 _productId, uint256 _claimId, uint256 _amount)
49      external
50      returns (uint256 _payoutId);
51
52  function payOut(uint256 _productId, uint256 _payoutId, uint256 _amount)
53      external
54      returns (uint256 _remainder);
55
56  function setPayoutState(
57      uint256 _productId,
58      uint256 _payoutId,
59      IPolicy.PayoutState _state
60  ) external;
61
62  function getApplicationData(uint256 _productId, uint256 _applicationId)
```

(continues on next page)

(continued from previous page)

```
    external
    view
    returns (
    uint256 _metadataId,
    bytes32 _customerExternalId,
    uint256 _premium,
    bytes32 _currency,
    IPolicy.ApplicationState _state
);

function getPayoutOptions(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (uint256[] memory _payoutOptions);

function getPremium(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (uint256 _premium);

function getApplicationState(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (IPolicy.ApplicationState _state);

function getPolicyState(uint256 _productId, uint256 _policyId)
    external
    view
    returns (IPolicy.PolicyState _state);

function getClaimState(uint256 _productId, uint256 _claimId)
    external
    view
    returns (IPolicy.ClaimState _state);

function getPayoutState(uint256 _productId, uint256 _payoutId)
    external
    view
    returns (IPolicy.PayoutState _state);
    }
```

CHAPTER 7

The query module

The **query module** allows any product contract to use oracles and access risk model data or get a confirmation about a particular real-world event off-chain.

The methods invoked by the query module include:

- **proposeOracleType** is called by oracle owners or product owners to submit a data input, a callback format, and definitions for a particular oracle type.
- **activateOracleType** is used by the instance operator to activate an oracle type.
- **deactivateOracleType** is employed by the instance operator to deactivate an oracle type.
- **removeOracleType** is used by the instance operator to remove an oracle type.
- **proposeOracle** is called by oracle owners or product owners to propose a particular oracle.
- **updateOracleContract** is called by oracle owners or product owners to update an oracle contract for a particular oracle.
- **activateOracle** is used by the instance operator to activate an oracle.
- **deactivateOracle** is used by the instance operator to deactivate an oracle.
- **proposeOracleToType** is called by oracle or product owners to propose a particular oracle to a specific oracle type.
- **revokeOracleToTypeProposal** is called by oracle owners or product owners to remove a proposal before it is approved.
- **assignOracleToOracleType** is used by the instance operator to assign an oracle to an oracle type.
- **removeOracleFromOracleType** is used by the instance operator to remove an oracle from an oracle type.
- **request** is called by a product to request data from an oracle by an oracle type.
- **respond** is called by the Oracle Service after an oracle response to respond to the request of a product.

Below, you can see how the above-mentioned methods can be invoked.

```
interface IQueryController {
function proposeOracleType(
    bytes32 _oracleTypeName,
    string calldata _inputFormat,
    string calldata _callbackFormat,
    string calldata _description
) external;

function activateOracleType(bytes32 _oracleTypeName) external;

function deactivateOracleType(bytes32 _oracleTypeName) external;

function removeOracleType(bytes32 _oracleTypeName) external;

function proposeOracle(
    address _sender,
    address _oracleContract,
    string calldata _description
) external returns (uint256 _oracleId);

function updateOracleContract(
    address _sender,
    address _newOracleContract,
    uint256 _oracleId
) external;

function activateOracle(uint256 _oracleId) external;

function deactivateOracle(uint256 _oracleId) external;

function removeOracle(uint256 _oracleId) external;

function proposeOracleToType(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external returns (uint256 _proposalId);

function revokeOracleToTypeProposal(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function assignOracleToOracleType(
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function removeOracleFromOracleType(
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external;

function request(
    bytes calldata _input,
    string calldata _callbackMethodName,
```

(continues on next page)

(continued from previous page)

```
58      address _callabackContractAddress,
59      bytes32 _oracleTypeName,
60      uint256 _responsibleOracleId
61  ) external returns (uint256 _requestId);
62
63  function respond(
64      uint256 _requestId,
65      address _responder,
66      bytes calldata _data
67  ) external returns (uint256 _responseId);
68      }
```

CHAPTER 8

The registry module

The **registry module** is responsible for registering sets of core contracts, which are used in a policy flow life cycle in release groups. The registry module is managed by the instance operator.

The functions available through this module are the following:

- **registerInRelease** is used to register new policies in a new release version.
- **register** is used to register a contract in the last release.
- **deregisterInRelease** is used to delete a contract from a release.
- **deregister** is used to delete a contract in the last release.
- **prepareRelease** is called to create a new release, move contracts from the last release to a new one, and update a release version.
- **getInContractRelease** is used to get a contract address depending on a release version.
- **getContract** is used to get a contract address in the last release.
- **getRelease** is used to get the last release's number.
- **registerService** is used to register a new service.
- **getService** is used to view a new service.

The code below illustrates how to invoke the functions of the registry module listed above.

```
interface IRegistryController {
function registerInRelease(
uint256 _release,
bytes32 _contractName,
address _contractAddress
) external;

function register(
bytes32 _contractName,
address _contractAddress
```

(continues on next page)

(continued from previous page)

```
) external;

function registerService(
bytes32 _name,
address _addr
) external;

function deregisterInRelease(
uint256 _release,
bytes32 _contractName
) external;

function deregister(
bytes32 _contractName
) external;

function prepareRelease(
) external returns (uint256 _release);

function getContractInRelease(
uint256 _release,
bytes32 _contractName
) external
view
returns (address _contractAddress);

function getContract(bytes32 _contractName
) external
view
returns (address _contractAddress);

function getService(bytes32 _contractName
) external
view
returns (address _contractAddress);

function getRelease(
) external view returns (uint256 _release);
}
```

CHAPTER 9

Use Cases for Product Owners

9.1 Register a product

For any product that expects to perform certain actions, it is crucial to register its product contract within the GIF instance.

The registration of a product contract takes place on a smart contract level.

A product creates a contract and inherits one of the GIF contracts — a product contract. After inheriting, a product contract is able to use the GIF functions to describe its business process.

The **register** function (see the code below) is used to register a new product contract. After approval, a contract obtains access to call entry methods.

```
function register(
bytes32 _productName,
bytes32 _policyFlow
) external
returns (uint256 _registrationId);
```

9.2 Role assignment by a product

To assign roles to specific contracts or people, role-based access control (RBAC) is used. A product contract should set up roles and specify what method can be called and by which role.

A lot depends on a business process, but there are two possible cases.

In the **first scenario**, actions can be called by people. It means a product contract may create a role, assign it to particular person, and this person will call a function (i.e., underwrite an application).

In the **second scenario**, actions perform a product contract. An oracle may respond with certain data, then a product contract will need to create a function of an oracle response handler, write certain logic, and automatically call the underwrite function.

A product owner defines necessary roles for its product contract and those who will be appointed to the created roles.

The data related to the roles is kept by a product contract. It inherits the "Product" contract and, after that, gets access to the RBAC methods. So, the roles belonging to the accounts and account data are stored in the product's account (without storage on the GIF).

The code below illustrates the contract details and function calls available.

```
contract RBAC {
        mapping(bytes32 => uint256) public roles;
        bytes32[] public rolesKeys;
        mapping(address => uint256) public permissions;
        modifier onlyWithRole(bytes32 _role) {
            require(hasRole(msg.sender, _role));
            _;
        }

        function createRole(bytes32 _role) public {
            require(roles[_role] == 0);
            // todo: check overflow
            roles[_role] = 1 << rolesKeys.length;
            rolesKeys.push(_role);
        }

        function addRoleToAccount(address _address, bytes32 _role) public {
            require(roles[_role] != 0);
            permissions[_address] = permissions[_address] | roles[_role];
        }

        function cleanRolesForAccount(address _address) public {
            delete permissions[_address];
        }

        function hasRole(address _address, bytes32 _role)
            public
            view
            returns (bool _hasRole)
        {
            _hasRole = (permissions[_address] & roles[_role]) > 0;
        }
    }
```

CHAPTER 10

Implementing a product policy workflow

10.1 Using a generic policy workflow

The GIF provides a list of entities to manage insurance business processes:

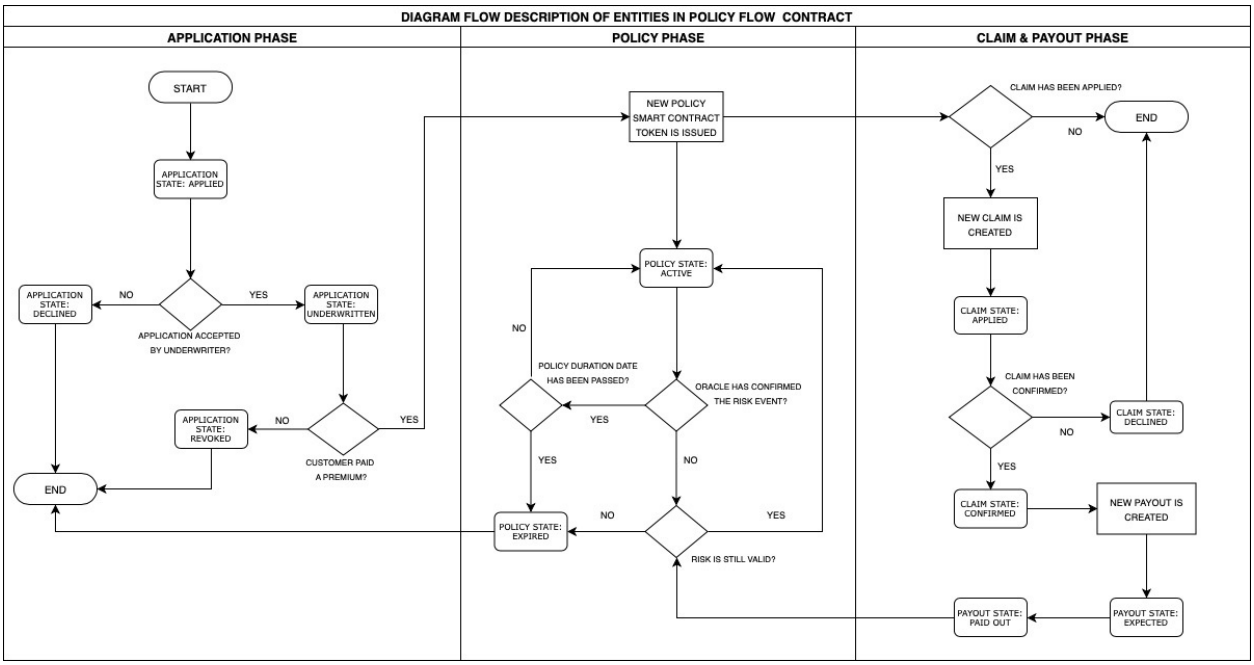
- an application
- a policy
- a claim
- a payout

These entities represent a generic policy workflow. In the course of a workflow, the state of entities will be changed visualizing insurance business process.

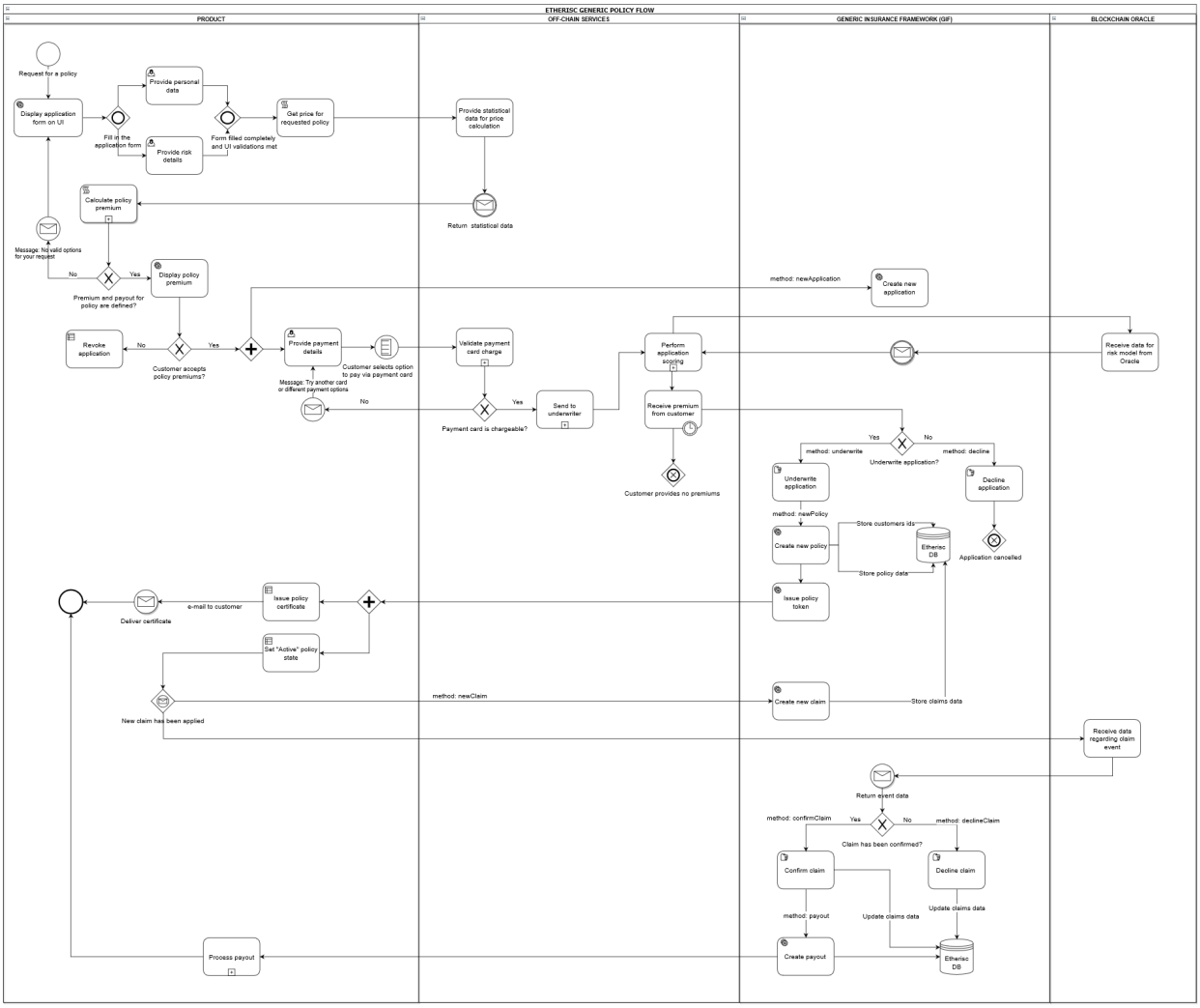
A product contract is able to use a workflow with both prepaid (before issuing a policy) and postpaid (after issuing a policy) premiums. On the diagram below, there are more details for a default scheme with prepaid premiums.

There are two possible ways of choosing premiums by a customer of a product: based on a fixed premium (a payout will correspond with a chosen premium) and based on a fixed payout (a premium will correspond with the desired amount of a payout).

A policy state flow diagram



The Business Process Model and Notation policy flow with prepaid premium diagram



Note: An example of a policy flow described above is one of the two possible flows (with a premium paid before a policy is issued). It is also possible to pay a premium after a policy issuance.

10.1.1 Managing an application

This insurance business process actually starts when any customer (or it might be an application from any organization that represents the interests of a group of customers) sends an application for an insurance policy via a user interface on a product contract level.

The DIP Protocol enables a product contract to perform the following actions:

- create an application
- underwrite an application
- decline an application
- revoke an application

Creating an application

To create an application for a policy, the **newApplication** function needs to be used. This function is invoked to store new application data, which contains such fields as a premium amount, currency, payout options, risk definition, etc. A policy buyer signs a policy agreement using this function.

The application state is "Applied." During scoring or underwriting processes, an application remains in the "Applied" status.

The code below demonstrates how the function is called.

```
function newApplication(
    bytes32 _customerExternalId,
    uint256 _premium,
    bytes32 _currency,
    uint256[] memory _payoutOptions
) internal returns (uint256 _applicationId) {
    _applicationId = productService.newApplication(
        _customerExternalId,
        _premium,
        _currency,
        _payoutOptions
    );
}
```

Underwriting an application

To sign a policy agreement by a product contract, the **underwrite** function has to be used.

As soon as an application is accepted by an underwriter, its state is changed to "Underwritten." The application remains in the "Underwritten" state when a new policy is created.

The code below demonstrates how the function is invoked.

```
function underwrite(uint256 _applicationId)
    internal
    returns (uint256 _policyId)
{
    _policyId = productService.underwrite(_applicationId);
}
```

Declining an application

This function is used simply to decline an application. The application state changes to "Declined."

The code below illustrates how the function performs.

```
function decline(uint256 _applicationId) internal {
    productService.decline(_applicationId);
}
```

10.1.2 Managing a policy

By default, before issuing a policy, an underwriter must confirm that policy premiums are fully paid.

When a customer has an application underwritten and paid a premium for a product policy, the GIF methods allow to fulfill the following actions:

- create a policy
- expire a policy

Creating a policy

This function allows to create a new entity: issue a new policy token. A policy is created with the "Active" state.

A product contract sends a PDF policy certificate to a customer using the PDF Generator core microservice.

Expiring a policy

The function is used to set a policy expiration. The possible cases are the following:

- A policy duration date has expired.
- A risk for a policy has been confirmed and paid out (in case a risk is to be paid out once).
- The event has not been confirmed by an oracle in the course of a policy duration, which means no payout.

When the function is performed, a policy state is set as "Expired."

The code below demonstrates how to use the **expire** function.

```
function expire(uint256 _policyId) internal {
    productService.expire(_policyId);
}
```

10.1.3 Managing a claim

The DIP allows products contracts to use the claim management methods. Specifically, the following actions can be performed:

- apply a claim
- confirm a claim
- decline a claim

Applying a claim

The function is used to declare a new claim. The claim state is set as "Applied."

Note: Claims can be applied when a policy has the "Active" or "Expired" status.

The code below demonstrates how the function is invoked.

```
function newClaim(uint256 _policyId) internal returns (uint256 _claimId) {
   _claimId = productService.newClaim(_policyId);
}
```

Confirming a claim

The function is used to confirm a claim. A new payout object is created after performing this action. The claim state is set as "Confirmed."

The code below illustrates how the function performs.

```
function confirmClaim(uint256 _claimId, uint256 _amount)
    internal
    returns (uint256 _payoutId)
{
    _payoutId = productService.confirmClaim(_claimId, _amount);
}
```

Declining a claim

This function is used to decline a claim. The claim state is set as "Declined."

The code below illustrates how the function is invoked.

```
function decline(uint256 _applicationId) internal {
    productService.decline(_applicationId);
}
```

10.1.4 Managing a payout

Confirming a payout

The method is used to confirm the payout that has actually happened. The payout state changes to "PaidOut."

```
function payout(uint256 _payoutId, uint256 _amount)
    internal
    returns (uint256 _remainder)
{
    _remainder = productService.payout(_payoutId, _amount);
}
```

10.2 Creating a new or update default policy workflow

Currently, the GIF offers a general purpose default policy workflow to products contracts. In case a product contract needs to update a default workflow or create a new one, there are three possible options to do this:

1. Pull a request from a product contract. This request will be reviewed by the Etherisc team and merged with the existing workflow. It may also be a new version of a policy workflow.
2. Create an issue on GitHub. A product contract can create an issue, and, after reviewal, the Etherisc team will plan the requested improvements on a policy workflow.
3. Direct a request via e-mail: contact@etherisc.com.

CHAPTER 11

On-chain and off-chain storage

Any Product on the DIP has a choice of:

- what type of data to store
- where to store data

The DIP storage model allows products contracts to store its data on:

- blockchain smart contracts
- a platform database
- a product database

Note: Payment card data should be stored on a payment provider level as it requires PCI compliance to store payment card data of customers in a database.

In many countries, a legal agreement is needed between a party that runs a storage service and a party that uses a storage service.

11.1 On-chain

As the DIP operates in the Ethereum environment, the term **on-chain** specifies smart contracts, where a product can store risk description and specific metadata per policy.

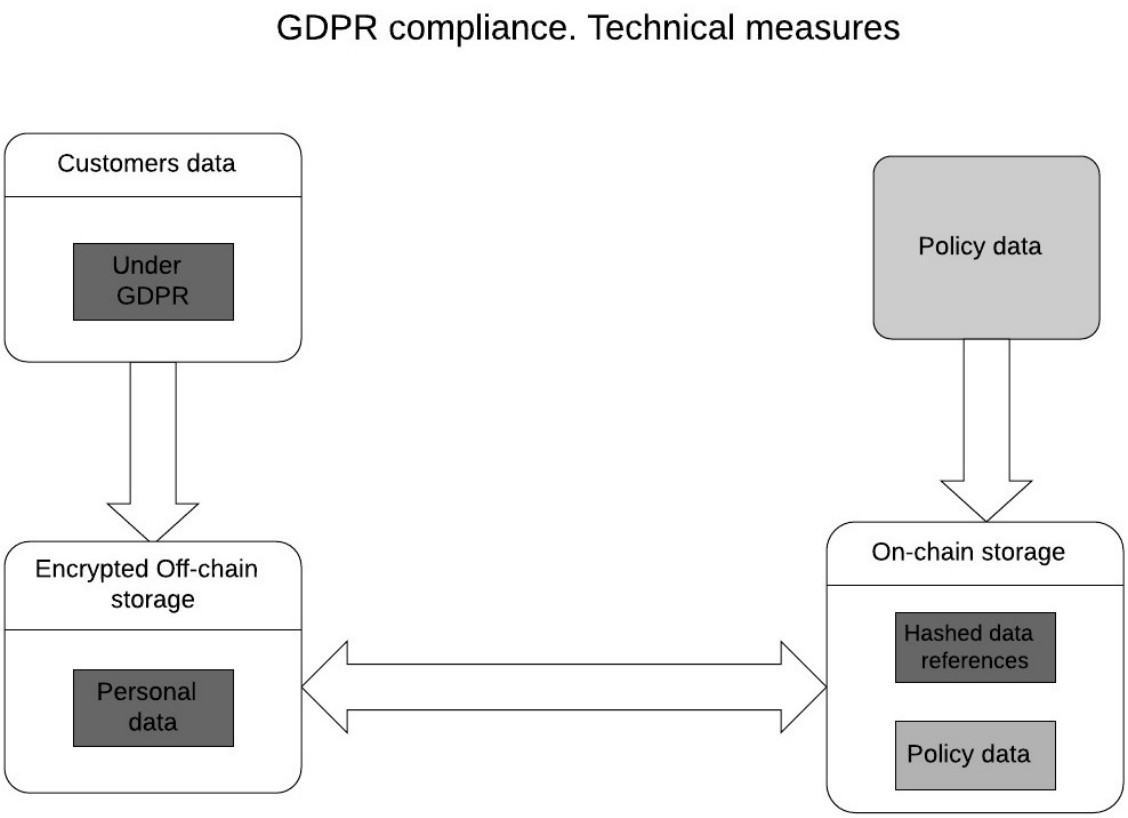
The key principle of how the DIP itself uses data is that no personal data is kept in smart contracts — only **unique hashed references**.

The DIP allows to store data for any product regarding its:

- customers (**Note:** *first_name, last_name,* and *e-mail* fields are required by the GIF.)
- policies
- claims

In case a product doesn't want to use a platform database, it is possible to use the product's database.

Attention: According to the EU General Data Protection Regulation requirements, we prevent you from storing personal data of customers on-chain. This data is to be stored **off-chain only**. There exist special identificators stored on-chain (hashed data references), which allow for retrieving data from an off-chain database. This prevents unauthorized access to sensitive data by an on-chain identificator. The diagram below illustrates the relations between on-chain and off-chain storage. This methodology implements the "Positionspapier des Bundesblock," the german association of blockchain companies which tries to implement this methodology in EU law.



11.2 Profiling

To avoid the possibility of the so-called customer "profiling," each newly issued policy gets a new unique customer ID (unique hashed reference).

CHAPTER 12

Make Payouts

In order to make payouts, a product contract can use the GIF **Payout** microservice. A product contract has two possible ways to be informed about the payout needed.

In the **first way**, a product contract can sign up to the **statusChanged** event from a policy storage to be notified when a payout is needed. The microservice gets a message from a smart contract and sends it to a product application, which needs to provide a payout.

The **second way** implies that a product contract can sign up to the **Event Listener** and read events about its contracts.

The GIF **Payout** microservice doesn't have any business logic implementation regarding where to transfer payout funds and which way (transferring to a bank account, payment cards, a transferwise, paypal accounts, coin wallets, post transfers, etc.). This business logic could be implemented by a product contract.

Below, you will find an example of a payout event, which has to be sent by a product contract.

```
{
policyId: 1,
payoutAmount: 100,
currency: 'EUR',
provider: 'transferwise',
}
```

CHAPTER 13

Managing oracles

13.1 Actors

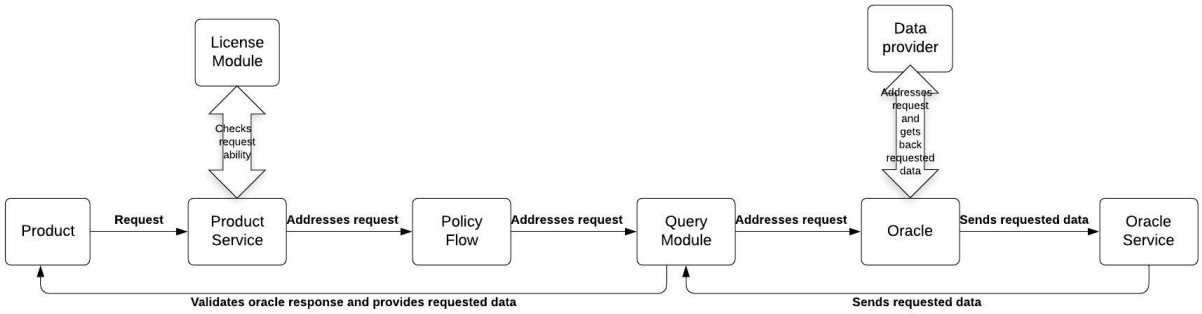
- A **Product** is a contract that provides a specific service to customers.
- The **Query module** is a service that forwards requests to different oracle providers.
- An **Oracle** is a service used to provide specific data to products.

13.2 Description

Oracle services provide huge leverage to the Generic Insurance Framework.

Product contracts use an oracle to obtain off-chain data and confirm or decline real-world events vital for an insurance process. The scheme below illustrates the request flow from the beginning (a product sends a request) till the end (a product receives the requested data).

The GIF accomplished a strategy for product contracts to get data from a specific oracle, which a product is particularly interested in.



13.3 A workflow

13.3.1 Registering an oracle

- Any product owner, oracle owner, or the instance operator is able to register its oracle type, where they specify criteria for the oracles that provide data back to the requesting parties. For this purpose, the **proposeOracleType** function is used. The parameters, such as *oracleTypeName*, *inputFormat*, *inputDefinitions*, *callbackFormat*, *callbackDefinitions*, and currency are defined here. Then, the instance operator activates an oracle type. It can also deactivate an oracle type.
- **deactivateOracle** is used by the instance operator to deactivate an oracle.

13.3.2 Registering an oracle to type

- An oracle owner, a product owner, or the instance operator can propose a particular oracle. For this purpose, the **proposeOracleToType** method is called to propose a particular oracle to a particular oracle type. The necessary parameters of the method are: *oracleTypeName*, *inputFormat*, *callbackFormat*, and a description.
- **revokeOracleToTypeProposal** is called by oracle owners or product owners to remove a proposal, before it is approved.
- **assignOracleToOracleType** is called by the instance operator to assign an oracle to an oracle type.
- **removeOracleFromOracleType** is called by the instance operator to remove an oracle from an oracle type.

13.3.3 Updating an oracle contract

- Oracle owners or product owners can update an oracle contract for a particular oracle.
- **updateOracleContract** is called to update an oracle contract.

13.3.4 Creating a request

- Product calls **request** a function.
- **ProductService** receives a request and verifies the correctness of the request according to the permissions of the License Module.
- **ProductSservice** addresses a request to a policy flow.
- A policy flow addresses a request to the query module to connect an oracle.
- The query module executes a request to a particular oracle that requests data from the data provider.
- An oracle calls to the requested data provider (i.e., Oraclize) to obtain the necessary data.
- **request** is called by a product contract to request data from an oracle by an oracle type. The request function uses the following arguments: *callbackMethodName*, *callbackContractAddress*, *oracleTypeName*, and *responsibleOracleID*.

13.3.5 Receiving a callback

- A particular data provider performs a callback to an oracle.
- An oracle sends a response to an oracle service with received data as an answer for the request.

- An oracle service addresses the received data to the query module, where the sender and the addressee are being verified. The query module specifies the response (which product contract made a query, what an oracle type is, and which Oracle is to respond to the query).
- Then, the response is to be checked. An oracle is confirmed to be registered to the system and to be assigned to an oracle type, which corresponds to that of the query. If everything matches, then an oracle provides a product contract with the requested data.

13.3.6 Making Respond

- An oracle contract makes a response using the **respond** method and sends the requested data in the **respond**.
- **respond** is called by an oracle service after an oracle responses to the request of a product contract.
- The methods of the query module are used to communicate with oracles when an insurance application requires data or a decision of a particular actor.

The code below illustrates the functions that can be called by the **OracleQueryController**.

```
interface IQueryController {

function proposeOracleType(
    bytes32 _oracleTypeName,
    string calldata _inputFormat,
    string calldata _callbackFormat,
    string calldata _description
) external;

function activateOracleType(bytes32 _oracleTypeName) external;

function deactivateOracleType(bytes32 _oracleTypeName) external;

function removeOracleType(bytes32 _oracleTypeName) external;

function proposeOracle(
    address _sender,
    address _oracleContract,
    string calldata _description
) external returns (uint256 _oracleId);

function updateOracleContract(
    address _sender,
    address _newOracleContract,
    uint256 _oracleId
) external;

function activateOracle(uint256 _oracleId) external;

function deactivateOracle(uint256 _oracleId) external;

function removeOracle(uint256 _oracleId) external;

function proposeOracleToType(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external returns (uint256 _proposalId);

```

(continues on next page)

(continued from previous page)

```
function revokeOracleToTypeProposal(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function assignOracleToOracleType(
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function removeOracleFromOracleType(
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external;

function request(
    bytes calldata _input,
    string calldata _callbackMethodName,
    address _callabackContractAddress,
    bytes32 _oracleTypeName,
    uint256 _responsibleOracleId
) external returns (uint256 _requestId);

function respond(
    uint256 _requestId,
    address _responder,
    bytes calldata _data
) external returns (uint256 _responseId);
}
```

CHAPTER 14

Upgrading policies

Once an insurance agreement is signed and a policy is issued, parties cannot unilaterally change the specified behavior. It means, the framework must ensure that all the parties involved can always exactly predict which set of smart contracts will execute the policy.

There are two ways to provide upgradability of contracts within the system and two different situations that could trigger an update of a smart contract:

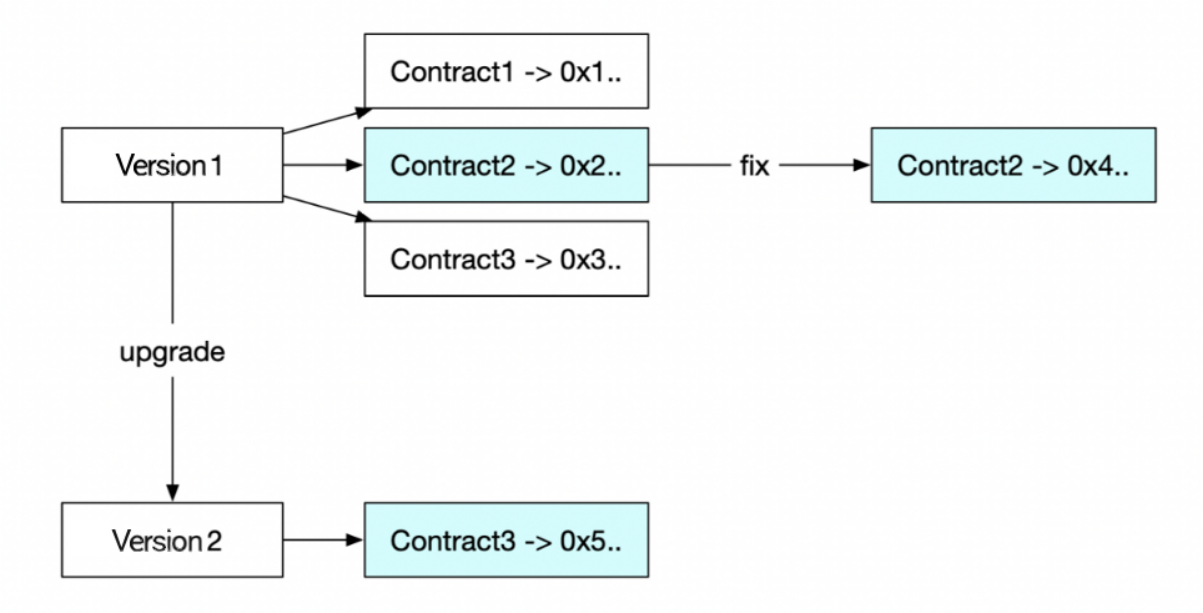
- **Fixes**. In case a bug has been detected. A bug is a deviation of expected behaviour to actual behaviour—a difference between a specification and an implementation. Bugs can be technical (a flawed implementation of a certain calculation leading to wrong results), but they can also occur in the translation process from legal prose to code. In this case, the specification would be flawed, and correct implementation of a flawed specification still leads to wrong results.
- **Upgrades**. New features need to be implemented for various reasons: modifications in pricing, a risk model, etc. In this case, the specification changes.

Both cases can occur on the core contract or product-specific contract levels.

We handle the two cases differently:

1. A "bug fix" upgrade will affect all policies, both the existing and new ones.
2. A "new feature" upgrade will affect only new policies. The existing policies will be executed with the original set of smart contracts (modulo bug fixes).

The following diagram illustrates the above-mentioned ways to change contracts and the result of such changes (when you get a new contract address). Then you can replace the existing contract by the new one if there was a "bug fixing," or deploy a new "additional" version of the contract if there was an upgrade of the existing contract.


Contract1 -> 0x1..
Version 1
Contract2 -> 0x2..
fix
Contract2 -> 0x4..
Contract3 -> 0x3..
upgrade
Version 2
Contract3 -> 0x5..