Implementation and Evaluation of an Incentivized Blockchain-Based Deposit-Refund System

Bachelor Thesis

for the

Bachelor of Science

in Applied Computer Science

at Baden-Wuerttemberg Cooperative State University Stuttgart

by

Niklas Sauer

September 3rd, 2017

Time of Project	12 Weeks
Student ID, Class	2677254, STG-TINF15A
Company	Hewlett Packard Enterprise
Location	Böblingen, Germany
Supervisor	Ralph Beckmann
Reviewer	Wolfgang Weyand
Manager	Ricardo Fernandez Diaz

Declaration of Authorship

Hereby I solemnly declare:

1. that this Bachelor Thesis, titled *Implementation and Evaluation of an Incentivized Blockchain-Based Deposit-Refund System* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Bachelor Thesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Bachelor Thesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Stuttgart, September 3rd, 2017

Niklas Sauer

Preface

This thesis presumes background knowledge in the realm of blockchain technology. At the very least, the concepts presented by Bitcoin should be familiar to the reader in order to ensure full comprehensibility of the technical proceedings presented and comparisons drawn herein. Published by the National Institute of Standards and Technology, [70] may be consulted as a general introduction to this landscape. In concert, the author's knowledgeability has been demonstrated through the research conducted in [55].

Abstract

Contents

Acronyms

ABI	Application Binary Interface
dApp	Decentralized Application
DPG	Deutsche Pfandsystem GmbH
EAN	European Article Number
EOA	Externally Owned Account
ERC	Ethereum Request for Comments
EVM	Ethereum Virtual Machine
HPE	Hewlett Packard Enterprise
NPM	Node Package Manager
QoS	Quality of Service

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Compared to glass, plastic or aluminum packaging represents a lightweight and durable alternative. The impact of lightweight materials on shipping costs is non-negligible and has therefore been leveraged in the beverage industry for the past 30 years. Simultaneously, the quote of reusable bottles (Mehrwegflasche) has steadily fallen (from 72% in 1991 [29, p. 1] to 43% in 2015 [30, p. 4]), which prompted German lawmakers to introduce a system of returnable one-way bottles (Einwegflasche) in 2003 on which a deposit is paid [57, p. 53].

Contrary to expectations [42, p. 10], this regulation has not stopped the influx of one-way bottles but has rather benefitted bottlers. Whenever consumers pollute by leaving behind one-way bottles, an instant 25 cent profit — assuming that no one else has returned them — is generated for the producer. This passive profit was estimated to have reached up to 192M€ in 2011 alone [51, p. 245].

Ideally, this pollution of the environment should be punished by splitting non-claimed deposits of one-way bottles between environmental agencies and those consumers who regularly purchase reusable bottles, which save more resources. As a further consequence, those consumers who repeatedly neglect to return their one-ways should be required to pay a higher deposit. Such a revised approach can hopefully maximize the number of returned one-way bottles and effectively steer users towards reusable ones. Otherwise, a further decline may be inevitable and is shown to have had a direct negative impact on global warming, in addition to the excess amount of waste produced hereof [2].

When considered on a case-by-case basis, this problem inherently deals with deposits of very low extrinsic value. Moreover, such a system has to manage account balances and track the movement of value, increasing the appeal and likelihood for external attacks. Therefore, implementing the proposed approach by utilizing smart-contracts and micro-transactions on the Blockchain may come to mind upon choosing the underlying infrastructure. But how does such an implementation look like; are there special considerations to be made?

1.2. Goals and Scope

Since no research on the feasibility of this approach has been undertaken for an application as specific as this one, the study focuses on the end-to-end development process, including design, implementation and deployment. The objective of the research is to guide Hewlett Packard Enterprise (HPE) and interested parties alike in developing a mindset suitable for migrating applications onto decentralized platforms and evaluate if this Blockchain-based implementation can withstand the requirements of an incentivized deposit-refund system. The results are relevant as they reduce the go-to-market time of the previously outlined solution to stop the influx of one-way bottles, which in turn reduces pollution and warming of the earth (comp. 1.1), a serious concern to today's society. Moreover, HPE can offer more profound Blockchain-related consulting services through the insights gained.

It shall be explicitly noted that the study does not cover analyzing the effectiveness of employing such a system (i.e. reducing the share of one-way bottles) or verifying wether this represents the best possible approach. Similarly, the legal framework and ethical questions pertaining to its usage are disregarded.

1.3. Thesis Overview

Chapter 2 reviews the relevant literature and is intended to answer all descriptive research questions that help define the project variables. The key concepts are then applied in chapter 3 as part of the architectural overview derived from the (non-)functional requirements of an incentivized deposit-refund system given earlier in the chapter. At the same time, these requirements will also act as the criteria used to evaluate the forthcoming implementation. Chapter 4 then describes the procedure to implement the solution after which the test-driven development approach will be outlined in chapter 5. Chapter 6 uses the results and obversations to draw a conclusion, discuss probable alternatives, as well as any limitations encountered during the study. Finally, chapter 7 summarizes the thesis's goals, methodology and results and is followed by chapter 8 to highlight interesting related research opportunities.

2. Theoretical Framework

2.1. Deposit-Refund System for Bottled Beverages in Germany

2.1.1. Legal Basis

Anticipating the depletion of capacity in disposal sites and due to the considerable share in waste generated by packaging [1] [42, p. 2], German federal government enacted an ordinance regarding the avoidance of packaging waste (*Verordnung über die Vermeidung von Verpackungsabfällen*, short: *Verpackungsverordnung - VerpackV*) in 1991 which stipulates that packaging [63, § 1]:

- must be minimized to an extent necessary for protection and marketing of goods
- must be reused where possible
- must be recycled if reuse is not applicable

These objectives were supported by introducing the:

Obligation to take back packaging

Producers and distributors of packaging are obliged to take back packaging free of charge, restricted to those goods of type, shape, size and material found within their stock [63, §§ 4-6]. Distributors with a retail area of less than $200m^2$ are further exempted to the same brands. This duty may be only be ignored if a distributor participates in a system which ensures the periodical collection of waste [63, § 6], known and implemented as a refuse recycling system (*duales System*) in 1990 [42, p. 3].

Obligation to levy deposits on beverage packaging [2]

A deposit is to be charged by the distributor that will be refunded to the purchaser upon return of the bottle. This duty is applicable on all levels of trade involving domestic beverages sold in non-reusable packaging (*one-way packaging*) [63, § 7] and

[1] Recycling rate of packaging was below 50% in the early 1990s (20% for aluminum and plastic) [42, p. 2].

[2] Information regarding the presumed effects and critique thereof can be found in [65, p. 630] and [64].

becomes effective as soon as the quote of reusable beverage packaging falls below 72% [3] [63, § 9].

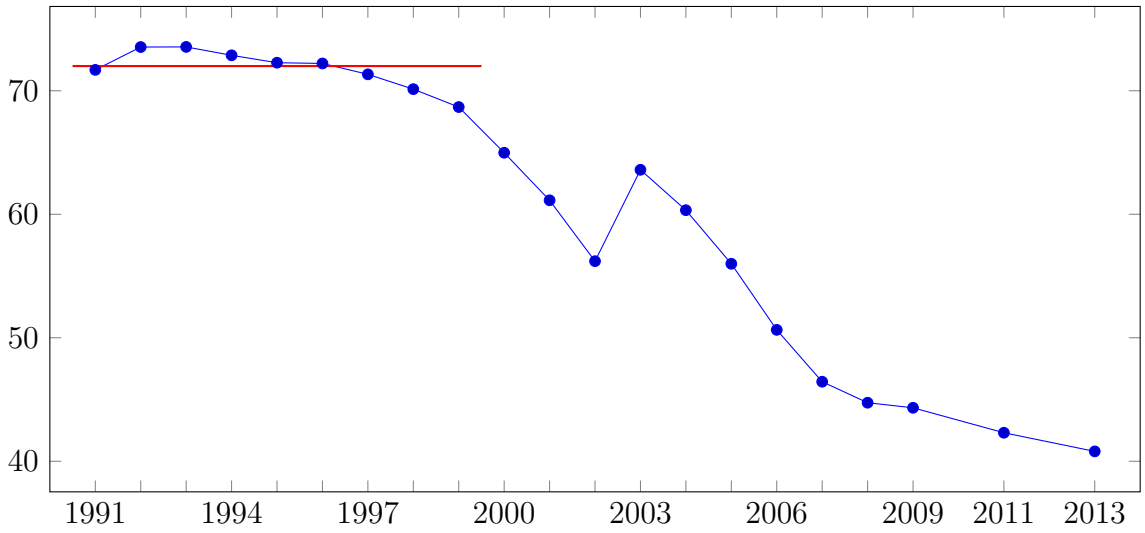

Figure 2.1.: Aggregated quote of reusable beverage packaging (in %) between 1991 and 2013 [29]

Starting in 1997, the threshold to levy deposits has been surpassed steadily (comp. Figure 2.1), requiring an additional assessment of the situation to ensure that the fluctuation does not represent a short-term development [63, § 9] [42, p. 5]. Although obvious at glance, the outcome of a compulsory deposit-refund system only became official on July 2nd, 2002 [57, p. 49], after a lawsuit lead by multiple bottlers and distributors had delayed the initial announcement [35]. Introduction of this mandatory system was scheduled for Jan 1st, 2003 [57, p. 53].

2.1.2. Amendments

The German packaging ordinance underwent several revisions, including a change of title to highlight the importance of recycling [62]. In the following, the most important changes affecting the current state (see Figure 2.2 on page 6) shall be highlighted.

1998

Trims deposit obligation to those beverages for which the quote of reusable packaging has fallen when compared to 1991, though still necessitates that overall aggregated quote undercuts threshold of 72% [38, pp. 142]. Even though all five segments have failed this comparison from 2000 onwards (comp. Table 2.1), wines, juices and non-carbonated soft drinks are exempted because their decline and market volume

[3] Aggregated quote derived from weighted average of percentages of reusable packaging encountered across individual segments in 1990 [63, § 9] [54, p. 134].

has not been regarded significant enough to justify the costs introduced with such a system [42, pp. 6, 9].

Segment	1991	1995	1998	2001	2004	2007	2013
mineral water	91.3	89.0	87.4	74.0	67.6	47.0	40.6
fruit juice & soft drinks	34.6	38.2	35.7	33.2	20.6	13.0	9.6
carbonated soft drinks	73.7	75.3	77.0	60.2	62.2	41.9	30.9
beer	82.2	79.1	76.1	70.8	87.8	85.2	85.9
wine	28.6	30.4	26.2	25.4	20.0	9.1	6.8

Table 2.1.: Quote of reusable beverage packaging per segment (in %) between 1991 and 2013 [29]

2005

Reduces different deposit classifications to single deposit worth 0.25 € valid for all applicable beverages with a filling volume between 0.1 - 3.0 liters. Furthermore, ecologically advantageous packaging is admitted the same treatment and classification as that of reusable packaging which represents an important shift of thought since the sole utilization of packaging has been considered inferior to its reuse with regard to all ecological aspects [42, p. 7]. Accordingly, the new goal is to promote the use of ecologically advantageous packaging with a target of 80% market dominance [40, § 1]. This amendment also adds non-carbonated soft drinks and mixed alcoholic drinks to the list of beverages subject to deposits (irrespective of the reusable packaging quote experienced) while simultaneously protecting dietary products from deposits [37, p. 1408] [39, p. 171]. Finally, retailers are required to take back any bottles made from a beverage packaging material (glass, metal, paper and plastic) also sold by that store (brand equality for retail areas of less than $200m^2$ still applies) [42, p. 11]. Previously, return had been limited to bottles of same shape, size [..] and type (comp. 2.1.1). This regulation attempts to stop isolated deposit-refund systems which arose because discounters created their own specially shaped bottles [39, p. 168]. To conform with this change, industry and commerce established the Deutsche Pfandsystem GmbH, responsible for setting up a nationwide deposit-refund system [42, p. 8].

2008

Orders distributors to clearly mark bottles as being obliged to a deposit. Further, distributors are demanded to participate in a nationwide deposit-refund system to enable the mutual settlement of claimed deposits [40, § 9]. Lastly, the exemption for dietary products is reduced to infant nutrition. This acts as a measure to counteract increasingly false product declarations attempting to forego deposits [40, pp. 531] [39, p. 171].

add 2019 amendment

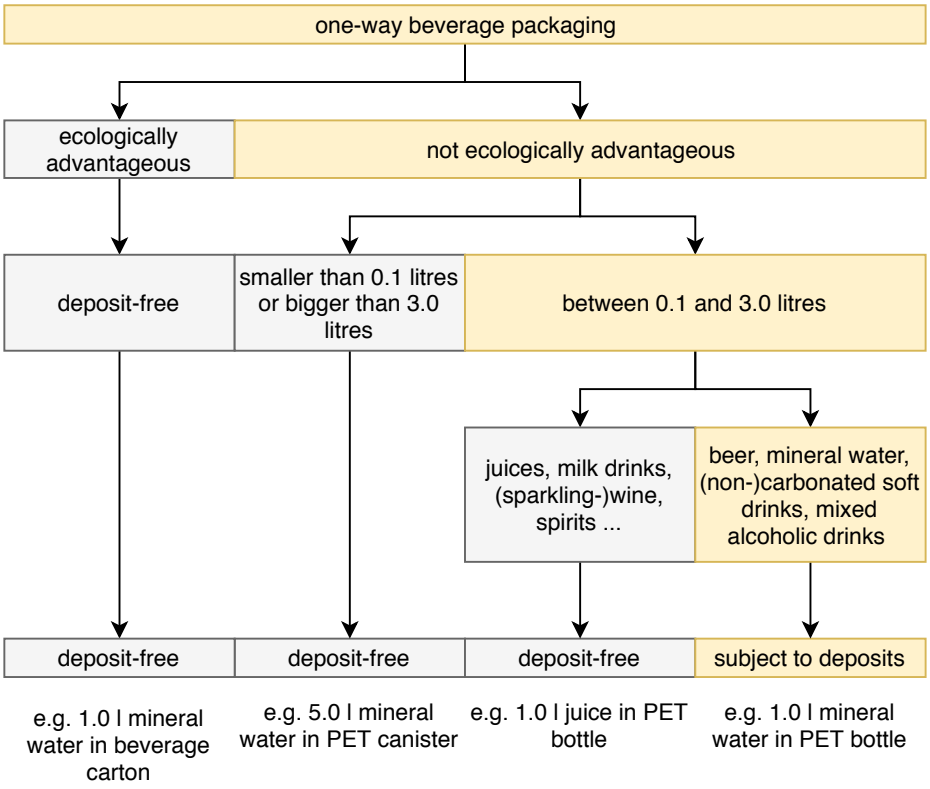

Figure 2.2.: Criteria to be considered for deposits (valid until Jan 1st, 2019) [42, p. 9]

2.1.3. Operation

Administration by Deutsche Pfandsystem GmbH

The German packaging ordinance simply assumes that a nationwide deposit-refund system will be installed by May 1st, 2006 without further specifying its exact implementation details [37, Art. 2] [40, § 9]. Therefore, Deutsche Pfandsystem GmbH (DPG) has been founded in 2005 to ensure the definition of a standards framework, which includes IT-processes, APIs and a universal security mark (see Figure A.1 on page 68) identifying one-way packaged bottles on which a deposit is paid (*returnable bottle*). Other tasks concern the management of contracts, certifications and maintenance of a centralized database which stores information about products, participants and reverse vending machines [42, pp. 13]. However, DPG does not have access to purchasing data (e.g. number of sold and returned bottles or total amount of reimbursed deposits). This data is only exchanged between actors of the deposit-refund cycle. Consequently, DPG is not involved in the process of settling refund claims between take-back points and does not act as a central clearing

house. It is exclusively responsible for providing a contractual and administrative basis needed to enable such a system [42, p. 14].

Roles

As previously indicated, actors of the deposit-refund cycle must register with DPG, accept the contractual basis for the role to be exercised and pay a yearly membership fee in order to participate. The most important roles are [42, pp. 15–16]:

Initial distributor
: Puts returnable bottle into circulation. Inherently tied to role of deposit escrow account manager. Must pay a one-time fee to register a beverage (i.e. an EAN) with DPG's central database. This fee depends on the number of pieces produced within a year for that beverage.

Deposit escrow account manager
: Administrates and disburses money received from levying deposit on returnable bottle. Task may be delegated to a service provider.

Take-back point
: Refunds consumer in exchange for returning bottles subject to deposits. Inherently tied to role of refund claimant.

Refund claimant
: Puts in a claim to be reimbursed for making advance refunds. Task may be delegated to a service provider.

Counting center operator
: Takes over role of reverse vending machine by confirming number of genuinely accepted bottles with a receipt.

Deposit-Refund Cycle

Each buyer of a returnable bottle must directly pay the designated deposit to the seller. This process starts with the initial distributor (step 1 of Figure 2.3) and is repeated for each sub-distributor until the bottles reaches a retailer which acts as the final distributor to end consumers (step 2 of Figure 2.3). After consumption, the consumer may return the packaging at any retail store obliged to take back the packaging in question (take-back point) (step 3 of Figure 2.3). Return may be performed manually (step 1a of Figure 2.4) or can be automated through reverse vending machines (step 1b of Figure 2.4). The former requires personnel to count the number of returned bottles and refund the consumer

accordingly. Following count, the collected bottles must be shipped to a counting center which is responsible for digitally capturing each bottle (step 2a of Figure 2.4). At last, the bottles will be compressed to eliminate the possibility of repeated refunds. In the case of reverse vending machines, the process of counting, capturing and compressing bottles can take place locally at a retail store (step 2b of Figure 2.4). Alternatively, retailers are permitted to return the packaging to the previous distributor, thereby unwinding the deposit-refund chain. However, this practice is rarely exercised since the packaging will not be reused and transportation would only induce additional ecological and financial strain. Instead, the compressed packaging can be sold directly to recycling companies (step 3 of Figure 2.4) [42, p. 16-17] [3].

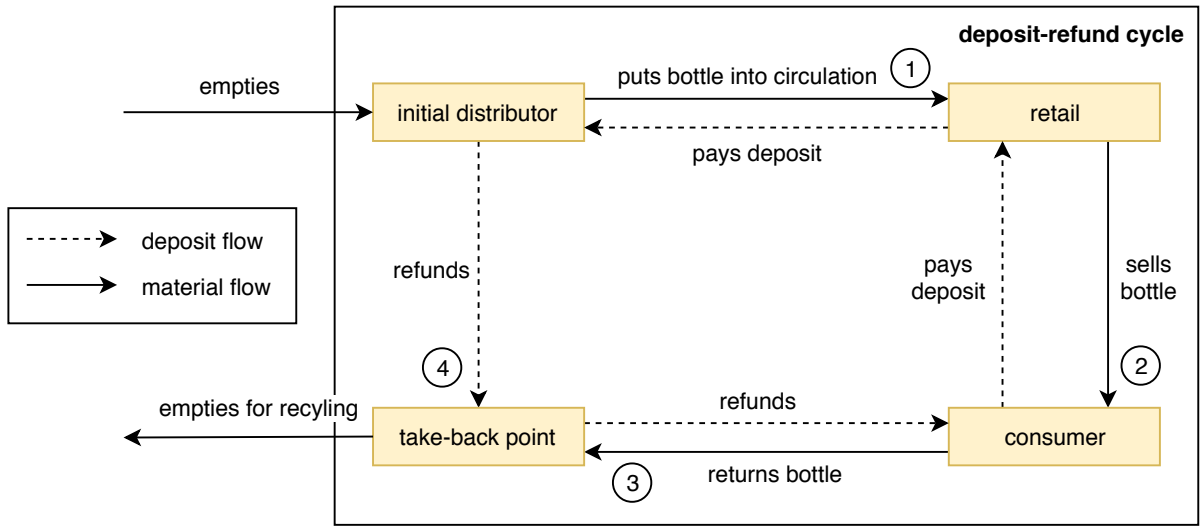

Figure 2.3.: Deposit-refund cycle [42, p. 14]

Since take-back points refund consumers independent of a bottle's purchasing location, a settlement process (*clearing*) is required to reimburse retailers for making advance refunds (step 4 of Figure 2.3). This clearing happens by issuing a refund invoice to the initial distributor who can use the deposits originally collected the pay the bill (step 4 of Figure 2.4). Ideally, all accounts are in balance, meaning that all packaging has been returned. Otherwise, a surplus on the part of an initial distributor is to be experienced [42, p. 17].

Refund invoices must also include data representing the number of bottles for which a refund was issued. This data originates from the return process in which each bottle is eventually captured. Capture refers to both validating the universal security mark printed on a bottle's label (see Figure A.1 on page 68) as well as extracting the European Article Number (EAN) by scanning its barcode. By outfitting reverse vending machines with a network connection, looking up the EAN in question becomes possible, thus ensuring that a deposit for the bottle has been collected beforehand. For each valid bottle, a signed

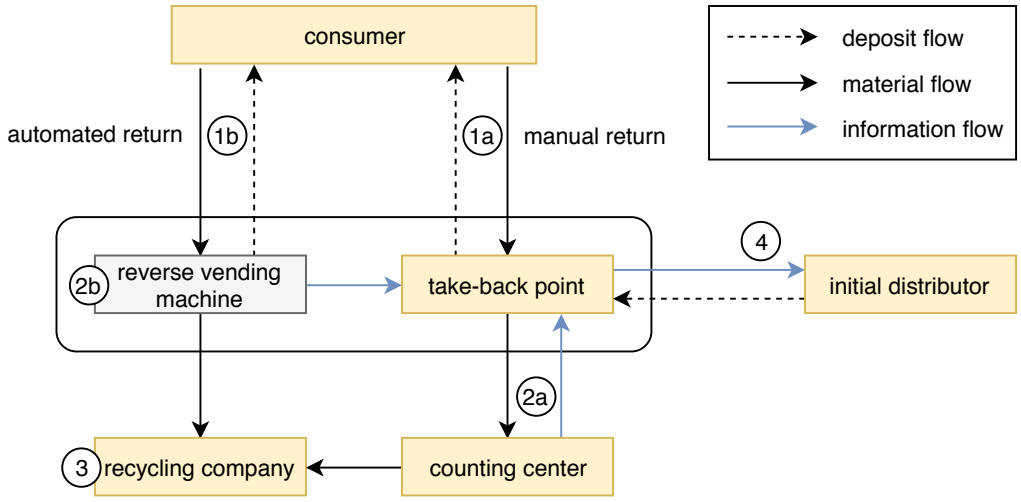

Figure 2.4.: Clearing process for manual and automated bottle returns [42, p. 18]

record is then produced and stored on the machine. This data must be retrieved and transmitted within one week. Likewise, retailers receive a similar dataset when relying on a counting center. Finally, any transmitted data is anonymized by removing the return location and summing up the refunds for each EAN captured. Initial distributors must pay the invoice within ten business days or five-teen calendar days at the latest [42, p. 18-19].

An extended diagrammatical overview of this cycle can be found in the Appendix A on page 69. It includes the various scenarios encountered nowadays which lead to differently balanced accounting states, as indicated previously. Also, the process of buying a returnable bottle from the perspective of a whole-seller or any other special entity such as a caterer is reflected in both figures since these parties are ultimately responsible for returning the one-way bottle as long as it has not been further distributed and by such, should be regarded as a consumer.

explain situation surrounding deposits on reusable bottles?

2.2. Decentralized Applications

2.2.1. History & Raison d'Être

Johnston's Law states that everything that can be decentralized, will be decentralized [1]. Fittingly, a new model for building massively scalable, successful and profitable applications, known as *Decentralized Applications (dApps)*, is emerging [53, p. 5] [43, pp. 1–2], suggesting that this will also become the fate of the vast majority of web software applications as those follow a centralized server-client model in which individuals directly depend on a central power to send and receive information [53, pp. 7–8].

The architectural structure embodied by the Web has not always been as pronounced as is these days. In its early days, people would host personal servers for others to connect to and everyone owned their data. But soon, it was recognized that one individual or group could pay for the maintenance of a server and profit from users that utilize it (e.g. Amazon Web Services [4]) and since this was easier to the average user, both conceptually and programmatically, the transition to few centralized controlling entities started [53, pp. 14].

Now, users willingly give their data to service providers in return for free usage, trusting them to not misuse or sell the data elsewhere [53, p. 24]. However, Snowden proved that this trust can, has and will be broken as long as we entrust (encrypted) data to central entities which represent a surveillance state's dream [27] [53, p. 25]. In addition, centralized infrastructure and services increase the chances for downtime, censorship and counterparty risk [25, p. 23]. Drawbacks like these have attributed to the development of dApps, first fully realized by Bitcoin [5] as a currency [43, p. 1] [48, p. 1]. Simultaneously, recent apps (e.g. Uber and AirBnb) attempt to decentralize real world parts of a business by providing a central and trusted data store, foreshadowing the development of decentralized apps beyond pure finances [6] [53, p. 15].

The concept of dApps is meant to take the web to its next [7] natural evolution [25, pp. 34]. Web3 represents the vision of a serverless [8] internet, encouraging applications to incorporate decentralized protocols in order to put users in control of their own data, identity and destiny [5]. As global economy evolves, data will become the primary form of value [53,

[4] Amazon controls roughly 40% of the cloud market and serves customers like AirBnb, Expedia, Netflix, Slack and Spotify [28]. Please install [4], to experience the impact of an AWS downtime.
[5] BitTorrent, as an earlier example, depends on centralized trackers for data discovery [53, p. 26].
[6] Supports the blockchain evolution theorem proposed in [59].
[7] Web 2.0 describes the evolution towards user-generated content, responsive interfaces and interactivity [25, p. 34].
[8] Not to be confused with the centralized Function-as-a-Service offerings detailed in [56].

p. 25], leading Raval to believe that "[dApps] will someday become more widely used than the world's most popular web apps" [53, p. 5].

2.2.2. Definition

Developers have different opinions on what exactly a dApp is or which components constitute it. Some think that no central point of failure is all it takes, whereas others name more specific requirements [53, p. 9]. The following will chronologically examine the various definitions available in order to derive one used for the remainder of this thesis.

2014

Buterin introduces the concept of a dApp as part of Ethereum's white paper. A complete dApp should consist of:

- low-level business-logic components
- high-level graphical user interface components

He goes on to state that "the blockchain and other decentralized protocols will serve as a complete replacement for the server for the purpose of handling user-initiated requests. ... Decentralized protocols ... may be used to store the web pages themselves" [32, p. 34].

This vague explanation is expanded through a consequent blog post (2014) in which Buterin formally defines the concept as being a superset of *smart-contracts* [31]:

Smart-contract
: Mechanism involving digital assets and a fixed number of parties, where some or all of the parties put assets in and assets are automatically redistributed among those parties according to a formula based on certain data that is not known at the time the contract is initiated.

Decentralized application
: Similar to a smart-contract but with unbounded number of participants and not necessarily of financial nature.

2015

To be considered a dApp, Johnston et al. assume that an application [43, p. 2]:

- is **open-source**, operates autonomously and adapts its protocol with user consent
- stores its data and records of operation in a **public blockchain**
- uses a **token** necessary for accessing it and rewarding contribution of value
- generates its tokens according to a predefined **cryptographic algorithm**

Moreover, he classifies dApps based on whether they have their own blockchain (Type I), use that of another dApp (Type II) or use a type II dApp as its underlying protocol (Type III). The latter is also known as a protocol [43, pp. 3–4].

2016

Although similar to the elements proposed priorly, the definition given in *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology* highlights their importance and by such requires that a profitable dApp [53, pp. 9–14]:

- is **open-source** to achieve transparency and gain trust among users
- achieves **decentralized consensus** on application-level constructs (e.g. high-score)
- uses an **internal currency** for access, reward and monetization
- has **no central point of failure** so that it cannot be shut down

2018

Finally, Antonopoulos and Wood (co-founder of Ethereum) back up and clarify Buterin's design guidelines by stating that a dApp is composed of at least [25, p. 34]:

- **smart-contracts** on a **blockchain**
- a web frontend **user interface**

Optionally, a dApp may rely on other decentralized components such as:

- decentralized storage
- decentralized messaging

Derived Definition

An application is considered to be a Decentralized Application (dApp) if it exhibits all of the characteristic features marked as a *must*. Secondary or recommended traits are labelled as *can* or *should* respectively.

It must be noted that this definition will not differentiate between type II and type III dApps because the former can be regarded as a dependency of the latter.

Open-source

Traditional business models require the service for sale to be better than that of a competitor. By open-sourcing a solution (both frontend and backend), copying becomes inevitable [53, p. 10]. However, users will want to remain with the team that is best suited to maintain the application [53, p. 11]. Therefore, developers should not fear trading secrecy for transparency. Transparency removes a barrier to adoption as users are no longer required to trust that an application works as advertised [53, p. 9]. Additionally, open-source development often sparks the attraction of enthusiastic developers who contribute freely [53, p. 11].

- *must* be deployed visibly (i.e. the executed code can be inspected by a user)
- *should* be developed on an open-source platform

Johnston et al. call for an application to only alter its protocol by the consensus of its users. Yet, it can be argued that this does not represent a core property of a dApp since users can revert to the desired state at any time by forking the project.

- *can* require the consensus of users to upgrade its protocol

Decentralized consensus

Usually, application-level constructs (e.g. usernames, high-scores or status updates) are managed and modified only through the centralized application provider, thereby ensuring a single and consistent state (*singleton*). Luckily, blockchains provide the means to reach consensus in a decentralized manner. Traditionally, this is only applied to transactional logs [48, p. 1]. Extending this consensus mechanism to cover the outcome of a computer program and consequently storing the result on a blockchain for future retrieval as an input, would enable virtually any application. Seeing that such an application would always have a single globally valid state, the underlying infrastructure can be colloquially termed a world-computer.

Raval, Antonopoulos and Wood, refer to these programs as smart-contracts and equally expect them to be secured by and executed on a blockchain [53, p. 13] [25,

pp. 23, 34]. It must be noted that Buterin consistently denotes smart-contracts as being a subset of contracts. The former is of financial nature and may involve redistributing digital assets among parties or unlocking value depending on the conditions met [32, pp. 1, 13]. The latter can be used to encode arbitrary state transition functions required to build decentralized applications which may also issue tokens [32, pp. 1, 19]. For simplification purposes, the business logic of both applications types will be referred to as smart-contracts.

- type I *must* achieve decentralized consensus on state-transitions
- type II *can* only modify its data via a smart-contract

Finally, in order to achieve decentralized consensus, it is ultimately necessary to have a complete picture of the state-transitions (*transactions*) which have been applied by a network. While theoretically possible, applications have never been observed to only store their transactions, due to the long processing needed to rebuild the most current state (*data*).

- *must* store its transactions in a public blockchain
- *should* store its data in a public blockchain

Internal currency

According to Raval, profit is a cornerstone of any successful, robust and sustainable application [53, p. 9]. Because traditional revenue streams (e.g. fees, advertising, subscriptions) are not applicable without preventing the possibility of a fork, he proposes an internal currency [9] that is required to use the service [53, p. 11], hoping that it will appreciate in value by being listed on an exchange. Nevertheless, not every application is designed for profit.

- *can* use an internal currency for magnetization
- *can* use an internal currency for access / service usage

On the other hand, using an internal currency to reward the contribution of value is especially important for type I dApps which employ their own blockchain and thus require an incentivization structure that keeps both miners and developers active [53, p. 9].

- type I *must* use an internal currency to reward the contribution of value
- type II *can* use an internal currency to reward the contribution of value

[9] Type I: Blockchain-native cryptocurrency; Type II: Smart-contract issued token.

Lastly, Johnston et al. specify that tokens shall only be generated according to a predefined cryptographic algorithm. This is inherently true for type I dApps which issue their internal currency as a reward for completing the cryptographic challenge (e.g. proof-of-work) [48, pp. 3–4]. In this context, tokens are considered to be a feature of type II dApps (comp. Decentralized consensus). Therefore, their generation is not directly tied to cryptographic algorithms, but rather to the program's control flow.

- type I *must* use a cryptographic algorithm to generate its internal currency
- type II *can* only generate its internal currency via a smart-contract

Decentralized protocols

Although often required, storing massive amounts of data in a blockchain defats its design purpose of a simple transactional log. Moreover, this would diminish the incentive to maintain the network because the cost to participate (i.e. disk storage and bandwidth) will become considerably higher than the reward paid if many applications follow this trend. Hence, other solutions should provide methods for storing data in a decentralized way that is robust and as trust-less as possible [53, pp. 25].

- *can* use a decentralized storage solution

Similarly, message-intensive applications should not (entirely) rely on a blockchain for operation. This is due to the fact that every message will be stored forever and is public by default.

- *can* use a decentralized messaging solution

The definition presented within this section purposely does not make any assumptions about the user interface or a dApp's resiliency. Command-line interfaces qualify just as much as mobile native apps, whereas resiliency can be maximized by employing additional decentralized protocols. More importantly, the primary focus and goal of any dApp developer should be to give users the confidence that a product works as promised by solely relying on open, decentralized peer-to-peer infrastructure services that can be inspected for functioning from end-to-end.

2.2.3. Tokens

By introducing a digital equivalent, blockchain has evoked a semantic change for the word token originally used to mean privately-issued coin-like items that have insignificant value and are designed for a single purpose (e.g. transportation-, arcade- or laundry tokens).

Now, tokens are used to simultaneously convey ownership, access rights, as well as the right to vote, for instance. Moreover, they are more easily exchangeable, a pitfall commonly cited [25, p. 173]. According to Antonopoulos and Wood, this could make tokens quite valuable when used in a way that does not re-create the conditions that made physical tokens worthless [25, p. 178], hereby hoping for industry wide solutions.

Use Cases

Antonopoulos and Wood present the following list of possible uses. However, tokens often encompasses several of these features and, much like their physical equivalents, it is hard to discern between them (e.g. driving license as identity and attestation) [25, pp. 173–174].

- Currency
- Resource
- Asset
- Access
- Equity
- Voting
- Collectible
- Attestation

Further, most projects employ tokens for two reasons: *utility* and/or *equity*. Similar to access tokens, utility tokens are those where the use of the token is required to pay for a service, application or resource [25, p. 176]. Yet, this requirement increases the barriers to adoption as the risks of broader economy (e.g. exchanges, liquidity, regulators) are added to that of the underlying blockchain technology [25, pp. 177]. Tokens should be adopted because the application cannot work without it, not because it presents a fast way to raise money while being disguised as a utility token to forego public securities regulations [25, p. 178].

Fungibility

Tokens are fungible if any single unit can be substituted for another without incurring a difference in value or function. A token is not entirely fungible if its historical provenance can be tracked, as this enables black- and whitelisting. The latter is the case for most cryptocurrencies [10]. Lastly, non-fungible tokens represent unique tangible or intangible items which are not interchangeable [25, p. 175].

[10] Monero pursues complete anonymity and therefore faces the possibility to be banned in Japan [24].

Intrinsicality & Counterparty Risk

While some tokens represent blockchain-native digital items, many tokens are used to represent extrinsic things. By such, consensus rules do not entirely apply and human law or agreements decide whether the transfer of ownership through tokens will be recognized. It is therefore even more important to understand who possess the associated asset and know what formal rules apply [25, pp. 175–176]. Leveraging tokens and smart-contracts to eliminate counterparty risk will most likely play part in future market strategies.

2.3. Ethereum

2.3.1. Overview

In line with the web3's vision of a serverless internet (comp. 2.2.1), Ethereum was designed to abstract the details of a blockchain to provide a deterministic and secure programming environment for dApps in which developers are no longer required to bootstrap the underlying mechanisms [25, p. 27]. By such, it is practically regarded as an open-source, globally decentralized computing infrastructure that executes programs, known as smart-contracts, for which the current state and transitions leading to this state are stored in a blockchain alongside a cryptocurrency called Ether. However, it must be emphasized that Ether is primarily intended as a utility currency [11] to meter and constrain resource usage during execution which stands in stark contrast with Bitcoin's sole purpose of being a digital payment network [25, p. 23] [48, p. 1].

Compared to a general purpose computer, the main differences are that [25, p. 28]:

- state changes are governed by rules of consensus
- state is distributed globally on a shared ledger

From a computer science perspective, Ethereum may therefore be described as a deterministic but practically unbounded state-machine characterized by its [25, p. 23]:

- globally accessible singleton state
- virtual machine that applies changes to that state

2.3.2. Developer Popularity & Adoption

Naturally, and as is the case with most base layer technologies, Ethereum does not represent the only smart-contract platform available [12]. Nevertheless, several key figures support the claim of Ethereum dominating this space:

- 94 of top 100 tokens (i.e. dApps) launched on Ethereum (87% of top 800) [61] [47]
- ranks well above competitors (#33 vs. #72 for EOS) when comparing traffic of dedicated sites in StackExchange network [6]

[11] Amusingly, Vitalik Buterin, co-founder of Ethereum, would like to use Bitcoin Cash for day-to-day transactions [68].

[12] Contenders include (alphabetically listed): Cardano, Corda, EOS, Hyperledger Fabric, Lisk, NEM, NEO, Stratis and Waves.

Moreover, Wang and Vergne argue that upon emergence of a new, technically more advanced platform, the blockchain with a stronger development team and open-source community is more likely to succeed [66, p. 14]. This developer and community focused nature is clearly seen within the Ethereum landscape:

- StackExchange topic encounters mainly developer focused questions [60, p. 6]
- ~14.3k answered questions (vs. ~850 for EOS) [68% vs. 81% of total questions] [6]
- ~17.5k repositories on GitHub (vs. ~3.4k for EOS & ~2.5k for Hyperledger) [13]

Most importantly, Ethereum is estimated to have around 250k developers [47], a 30-fold lead to Hyperledger Fabric, the second most active community, according to a report from Gartner [34].

Similarly, the degree of decentralization should not be neglected when considering platforms based on public permissionless blockchains. To this extent, Ethereum boosts ~17.6k nodes across six continents [14] [9].

2.3.3. Concepts

In order to provide developers with a general-purpose computing architecture that runs programs everywhere, yet produces a common state secured by the rules of consensus [25, p. 31], Ethereum redefines existing concepts, while also introducing several new ones.

Accounts

At its core, the state of Ethereum is composed by objects called *accounts*, each of which has its own 20-byte address and features the following properties [67, p. 17]:

- Nonce
- Current Ether balance
- Contract code (if present)
- Storage (empty by default)

Further, two types of accounts are distinguished [67, p. 17]:

[13] A fuzzy, top-level search has been conducted on GitHub on August 15th, 2018.

[14] EOS does not qualify since the 21 block producers used in its delegated proof-of-stake consensus mechanism are elected from a list of pre-approved candidates [41, p. 6] [7]. A similar case can be made for NEO which is only currently in the process of decentralization [49] [8].

Externally owned accounts (EOAs)

Accounts created by or for human users of the Ethereum network. This implies the existence of a private key which controls access to the associated funds [25, p. 13]. Such an account has no code, but can send messages by creating and signing a transaction.

Contract accounts

Accounts containing code that executes whenever a message is received from another account [25, p. 13]. Consequently, they are owned and controlled by the logic of the code stored respectively [25, p. 57]. This code allows contract accounts to read and write to internal storage and send messages. The latter unlocks additional capabilities such as transferring Ether and creating new or interacting with existing contracts [67, p. 19].

Messages & Transactions

As previously indicated, a transaction is a signed data package that stores a message which originated from an EOA. It works as expected from any cryptocurrency (i.e. it results in a transfer of value, if confirmed and non-zero, positive amount was specified) but has been expanded by three additional fields for contract invocation [67, p. 18]:

- recipient
- signature (identifies sender)
- amount of Ether (denoted in Wei [15])
- *data* (optional)
- *start gas* (required)
- *gas price* (required)

On the other hand, messages are virtual objects that are never serialized [67, p. 19]. This is due to the fact that all messages can be reconstructed by replying the transaction that triggered a contract's code to run and thus, produced the message in question [16]. More, they only implicitly contain the sender's address and do not allow the gas price to be specified as is explained in the following section.

Gas

Whereas Bitcoin's scripting language is constrained to simple true/false evaluation of spending conditions, Ethereum's language is Turing-complete, meaning that it can execute

[15] Smallest unit of currency, from which 1 Ether $= 1 \times 10^{18}$ Wei. Following the International System of Units, Ether's denominations have both a scientific name, as well as a colloquial name that pays homage to the great minds of computing and cryptography [25, p. 40].

[16] Messages are generated whenever a contract uses the `CALL` upcode [67, p. 19]

code of arbitrary and unbounded complexity [25, p. 25]. In turn, this leads to additional security and resource management problems as it cannot be predicted whether a program will terminate or run in infinite loops to effectively cause a Denial-of-Service. To combat this attacking scheme, Ethereum introduces a metering mechanism called *gas* which accounts for every instruction performed and assigns each a predetermined cost in units of gas [17] relative to the degree of computation required or burden imposed from writing to a persistent data store [25, pp. 32–33]. For simple transactions (i.e. a payment), the amount needed has been fixed at 21k units [25, p. 153].

By requiring transactions to set an upper limit (`STARTGAS`), execution will deterministically terminate as soon as the gas consumed exceeds the gas supplied [25, p. 33]. This gas allowance applies to both the transaction itself, as well as all sub-executions triggered through the encapsulated message [67, p. 19]. Logically, the total amount to be consumed by invoking a particular method can only be estimated because contracts can evaluate different conditions leading to different execution paths. In any case, EOAs are only billed for the gas actually used [18] [25, p. 154].

Additionally, the `GASPRICE` field allows an originator to set the exchange rate for each unit of gas (measured in Wei per gas unit). Analogous to other cryptocurrencies, the higher the gas price [19], the faster the transaction is likely to confirm. Still, it may also be set to zero and the transaction might even get mined during periods of low demand [25, p. 153].

With these two parameters in place, the final transaction fee, collected by miners, can be calculated as specified in Equation 2.1 [25, p. 53] [67, p. 20].

$$\text{fee} = \text{consumed gas} \times \text{gas price} \tag{2.1}$$

In this sense, gas is the fuel of Ethereum and has been purposefully separated as such to protect the system from Ether's volatility in value [25, p. 152]. At the same time, developers need to think of incentives for people to use the application since invocation of a smart-contract ultimately incurs monetary costs [60, p. 4].

[17] The current fee schedule can be found in [69, p. 25].
[18] Assert-style exceptions consume all gas available [58, p. 75]
[19] Often, the suggested gas price is calculated as the median across the last several blocks [25, p. 153]. However, gas prices are predominantly determined by miners and can therefore fluctuate unexpectedly [25, p. 54].

2.3.4. Functioning

State Transition Function

Like any (distributed) state-machine, Ethereum requires a function to transition the current state (s) to the next final state (s'). This function is defined as $apply(s,\ tx) \rightarrow s'$, where tx represents an unprocessed (i.e. unconfirmed) transaction. Application works as following [67, p. 20]:

1. Check if transaction is well formed, contains valid signature and matches nonce
2. Calculate fee = start gas × gas price, determine sending address from signature, subtract fee and increment nonce
3. Set gas = start gas and take off certain quantity per byte to pay for in transaction
4. Transfer value to receiving account and run specified contract code if present
5. If transaction failed due to insufficient funds or out of gas exception, revert all state changes except payment of mining fees
6. Otherwise, refund remaining gas to sender and add fee to miner's balance

Code Execution

Instead of only tracking the state of currency ownership, Ethereum tracks a general-purpose, key-value data store [25, p. 28]. More complex state transitions (i.e. not a payment) can be achieved by loading the code stored for a specific contract account into memory and running it on an emulated computer called the Ethereum Virtual Machine (EVM) [25, p. 57]. EVM code is written in a low-level, stack-based bytecode language, generated from compiling one of the many high-level languages available [20] (e.g. LLL, Solidity, Vyper) [67, p. 22] [25, p. 29].

In general, code execution is an infinite loop that repeatedly carries out the operation at the current program counter until the end of code is reached or an `ERROR`, `STOP` or `RETURN` instruction is detected. During execution in the EVM, code may access the value, sender and data of an incoming message, in addition to basic block header data and the global account state. Furthermore, operations have access to three types of space in which to store data [67, p. 22]:

- stack (follows LIFO-principle with push and pop methods)

[20] A curated list of compatible smart-contract languages can be found at [10].

- memory (infinitely expandable byte array)
- storage (persistent key-value store)

Lastly, it must be mentioned that the EVM is modeled as a completely isolated, sandboxed process preventing access to the network and filesystem [58, p. 19]. This has the negative effect of rendering most traditional APIs useless [53, p. 46].

Block Validation

Ethereum, by definition, is a system that allows concurrency of operations but enforces a singleton state at each mined block [25, p. 151]. To ensure that this state (also applies to forks) represents a valid state, the following block validation algorithm must be carried out by each node upon downloading a foreign block [25, p. 57] [67, pp. 23]:

1. Check if previous block referenced exists and is valid
2. Check that timestamp is greater than previous but less than 15 minutes in future
3. Check that block number, difficulty and various other low-level Ethereum-specific parameters are valid
4. Check that proof-of-work on block is valid
5. Set $s[0] =$ state of previous block
6. Set $tx =$ block transaction list (with n transactions). For each i in $0 \dots n-1$, set $s[i+1] = apply(s[i], tx[i])$. Return an error, if any iteration returns an error (comp. State Transition Function) or total gas consumed exceeds the `GASLIMIT` [21].
7. Set $s_{final} = s[n]$ with block reward paid to miner
8. Verify that Merkle tree root of s_{final} is equal to root provided in block header

This procedure also directly answers the question where contract code will be executed in terms of physical hardware. Namely, on all nodes that download and validate blocks containing transactions whose data field matches one of the methods of the recipient's contract code [67, p. 24].

[21] Similar to Bitcoin's block size, the `GASLIMIT` regulates the maximum number of transactions that can be included within a block [11] [69, p. 8]. But instead of solely measuring storage space, it limits the amount of computation that a miner must apply.

3. Concept

3.1. Incentivized Deposit-Refund System

3.1.1. Overview

An incentivized deposit-refund system, as is outlined in the following, aims to:

(A) bolster the quote of beverages sold in reusable packaging

(B) prevent pollution of the environment caused from throwing away bottles

(C) support the cause of environmental agencies

From a game theoretical perspective, (A) and (B) are presumably achieved by [22]:

- rewarding consumers who purchase reusable bottles
- punishing consumers whose bottles are eventually thrown away

To translate these measures into reality, it will be necessary to alter the current deposit-refund system (see 2.1.3) so that:

- non-claimed deposits are redistributed among agencies and reusable bottle consumers
- a penalty is added onto future deposits of consumers who repeatedly pollute

Practically, this will require garbage collection to report how many and which specific one-way bottles have been thrown away. Otherwise, it will not be possible to calculate the amount of non-claimed deposits and identify which consumer was responsible for the bottle at last. The latter infers that bottles must be uniquely identifiable. On the other hand, such a system can only be realized if retailers forward the information that a consumer has bought reusable bottles and has thereby become eligible to receive rewards. Furthermore, to establish proper ownership, they must also report which one-way bottles a consumer has purchased. Lastly, it should be prevented that anybody can report these figures, pretend to be an environmental agency or claim someone else's rewards. This core

[22] Lending itself to the fact that solely threatening retailers and bottlers with the introduction of a deposit-refund system had not been a successful approach (comp. footnote 2 and Figure 2.1 on pp. 3–4).

functionality, together with the existing minimum level of service, can be expressed in terms of the different interactions a certain user may have with the system, known as a use case diagram and given by Figure 3.1.

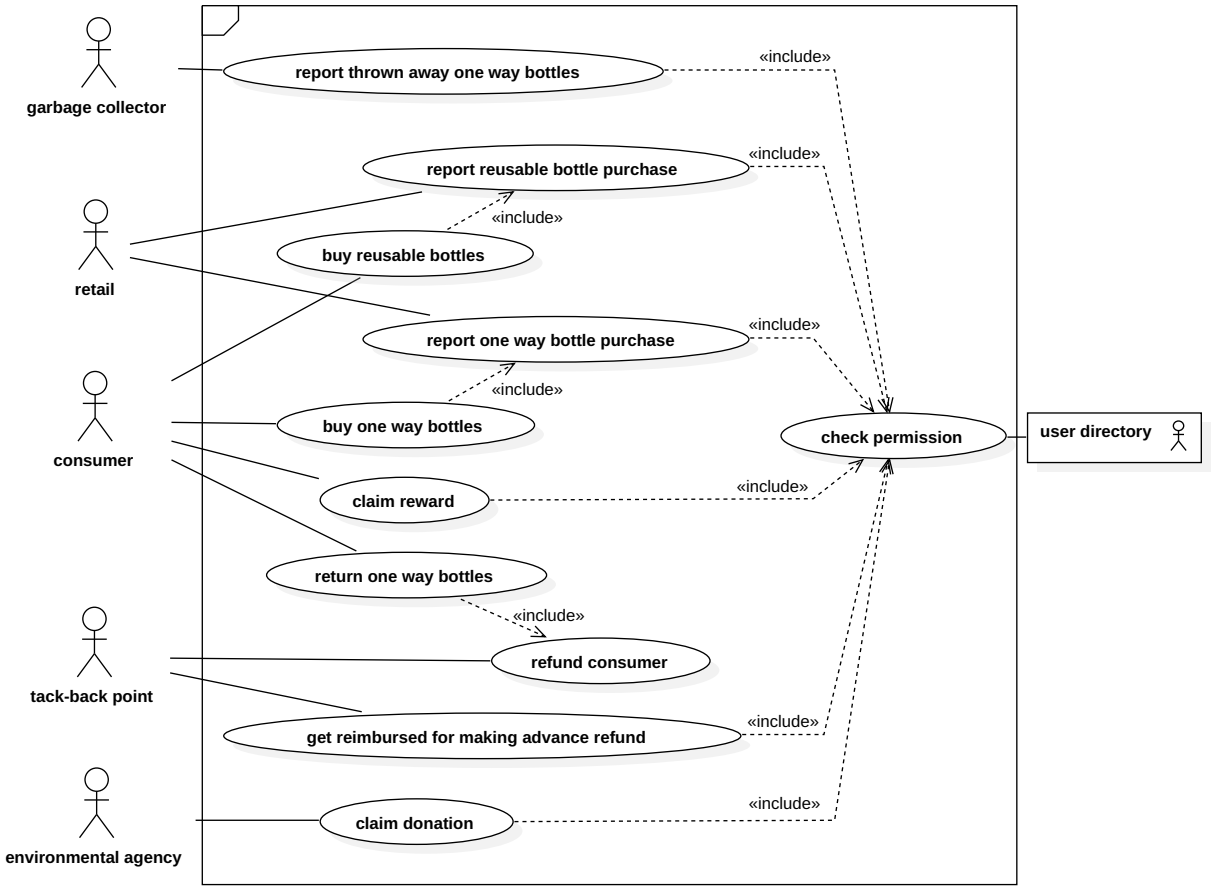

Figure 3.1.: Use case overview

From this overview, it becomes apparent that a revised approach will essentially incorporate an accounting layer to enable the punishment and rewarding of consumers. Basic protocols of interaction (i.e. buying and returning a bottle) should only be modified to the minimum extent needed.

3.1.2. Proposed Rules

Having given a high-level introduction to an incentivized deposit-refund system, this section aims to provide an exemplary set of rules that may constitute the underlying business logic. Again, it must be noted that this only serves as a guideline for architecting such a solution. The precise economics, psychological effectiveness and ethics introduced hereby are disregarded (comp. 1.2).

Rewards & Donations

- 50% of non-claimed deposits are reserved for donations (*agency fund*)
- 50% of non-claimed deposits are reserved for rewards (*consumer fund*)
- rewards and donations can be claimed once in each period
- a period's duration is set at 4 weeks
- a consumer's reward (r_c) is based on the relative amount of reusable bottles (b) he has purchased within the previous period, thus can be calculated as:

$$r_c = \frac{b}{b_{total}} \times f_c \tag{3.1}$$

  where b_{total} denotes the total number of reusable bottles sold within that period and f_c resembles the consumer fund
- non-claimed rewards are withheld
- the agency fund (f_a) is evenly distributed, so that each's donation (d) will be $d = f_a/n$, where n refers to the number of approved agencies
- non-claimed donations remain available for all agencies in the next period

Penalties

- if a consumer repeatedly pollutes, he must pay a penalty on top of each deposit
- the threshold (t) to receive a penalty is set at 5 thrown away bottles
- the penalty's value (v) is set at 0.05 €
- a consumer's penalty (p_c) is proportional to the number of bottles (b) he has thrown away during the lifetime of this system, meaning that:

$$p_c = \frac{b - (b \mod t)}{t} \times v \tag{3.2}$$

- the penalty is refunded, if the consumer returns the bottle on his own
- the penalty is seized, if the bottle is thrown away or returned by someone else

3.1.3. Functional Requirements

In accordance to subsection 3.1.1 and subsection 3.1.2, the following functional requirements are defined as the bare minimum of features any implementation must provide in order to be used as the accounting backend of an incentivized deposit-refund system. These requirements are expressed from the perspective of an end-user who hopes to achieve different goals by using the system and are grouped by the various roles encountered (comp. Figure 3.1).

Of course, consumers and environmental agencies will also expect a method to claim the promised rewards and donations respectively, so that two additional requirements are formulated subsequently.

Garbage collector

FR-01 As a garbage collector, I can report the number of thrown away one-way bottles, so that the amount of non-claimed deposits can be calculated.

FR-02 As a garbage collector, I can report the identifier of a thrown away one-way bottle, so that the responsible consumer can be punished.

Retailer

FR-03 As a retailer, I can report that a consumer has purchased reusable bottles, so that he becomes eligible to receive rewards.

FR-04 As a retailer, I can report how many reusable bottles a consumer has bought, so that his share of rewards can be calculated.

FR-05 As a retailer, I can report which one-way bottles a consumer has bought, so that the ownership of each bottle is recorded.

FR-06 As a retailer, I can look up the penalty that a specific consumer must pay, so that this amount can be added onto the deposit of each bottle during purchase.

Take-back point

FR-07 As a tack-back point, I can report which one-way bottles a consumer has returned, so that the penalty, if raised, can be refunded or seized accordingly.

Consumer

FR-08 As a consumer, I can claim a reward for having purchased reusable bottles, so that I will be motivated to so in the future even though one-way packaging is lighter (and more durable) when compared to most reusable bottles.

Environmental agency

FR-09 As an environmental agency, I can claim a donation, so that my cause will be better supported through the funding secured.

3.1.4. Preconditions

While an incentivized deposit-refund system may be effective, it does assume a variety of conditions that must be in place for it to function properly. As previously hinted, they can be described as the:

Obligation to report thrown away one-way bottles

In addition to reporting the total number of thrown away bottles, garbage collection must also be obliged to communicate the individual identifiers. This assumes that one-way bottles are marked with a label that is unique across all bottles in circulation. Finally, and to automate the whole process, special object recognition machinery will be required along the sorting belt of any given refuse disposal site.

Obligation to report bottle purchase

Retailers and initial distributors alike must report which particular one-way bottles a consumer has purchased. The same (i.e. quantity) goes for buying reusable bottles although this only happens through opt-in as people should not be forced to accept rewards. In both cases, it will be necessary to have consumers present a unique means of identification as they complete the purchase.

Obligation to levy penalty

Assuming that the consumer qualifies for a penalty, retailers must be impelled to impose this amount upon purchase which itself requires an extension to cash registers so that the amount can be queried dynamically, given that the consumer has been identified priorly.

Obligation to report one-way bottle return

Similar to reporting a purchase, tack-back points have to report which consumer had been responsible for returning a particular one-way bottle. Since the return

process can either be automated or executed manually (comp. Figure 2.4), different mechanisms to identify the consumer will be needed.

These prevailing conditions could be enforced by amending the system's legal basis (see 2.1.1) and then integrating the relevant clauses into the contracts that DPG maintains with each participant (see 2.1.3). However, the exact realization including that of the measures needed as explained above is out of scope (comp. 1.2) and thus, assumed to be satisfied for the remainder of this paper.

3.1.5. Non-Functional Requirements

Whereas functional quality stresses conformance with the design specifications, structural quality addresses non-functional requirements like security and maintainability [46, p. 2]. Together, both can be used to constitute the evaluation framework for measuring a system's quality, better known as Quality of Service (QoS). Liu, Ngu, and Zeng argue that it is not practical to come up with a standard model of attributes because QoS is a broad, context-dependent concept [44, p. 67]. Therefore, the following list of desired structural properties is based on those already encompassed in the current deposit-refund system or those which are absolutely necessary to implement an incentivized version. All other traits will be left open for discussion in the end.

Access Security

NFR-01 The system shall distinguish between authorized and non-authorized users. More specifically, the individual features outlined above shall only be accessible to their designated user.

Availability

NFR-02 The system shall be available for use between the hours of 6:00 am and 24:00 pm which is justified by the fact that most sales and returns will happen within this timeframe. Clients must therefore maintain a backlog of unsent reports and transmit these exceptions at the next possible point in time.

Cost

NFR-03 The system shall be no more costly in operation than it currently is. Total membership and bottle registration fees may be taken as a baseline (see 2.1.3).

Scalability

NFR-04 The system shall be able to handle all user requests and by such, shall be scalable to support unlimited growth in the number of actors, reports and claims.

Efficiency

NFR-05 The system shall execute any request by the end of the day to ensure the timely application of penalties, as well as to have a proper statement ready by the end of the month.

Integrity

NFR-06 All monetary amounts (i.e. rewards and donations) must be calculated accurately before being rounded down to two decimal places. The latter ensures that no more funds are distributed than are available. Additionally, only integer figures may be reported and must be recorded as is.

3.2. Architecture

3.2.1. Smart-Contract as a Clearing House

The biggest roadblock to implementing an incentivized deposit-refund system (see 3.1) is given by the fact that DPG has decided against using a central clearing mechanism to settle refund claims of tack-back points (comp. Administration by Deutsche Pfandsystem GmbH on page 6). Instead, refund claimants must invoice each of the initial distributors for which a bottle was taken back and present data proving the number of accepted bottles (comp. Deposit-Refund Cycle on page 7). This design choice is easily explained because the law mandates that: "a deposit is to be charged by the distributor and each subsequent retailer" (comp. Legal Basis in 2.1.1). Obviously, transferring the deposit to an escrow account afterwards would not lead to any noticeable benefits compared to direct a settlement. Still, such a change in course will represent the major enabler to an incentivized deposit-refund system as is depicted in Figure 3.2.

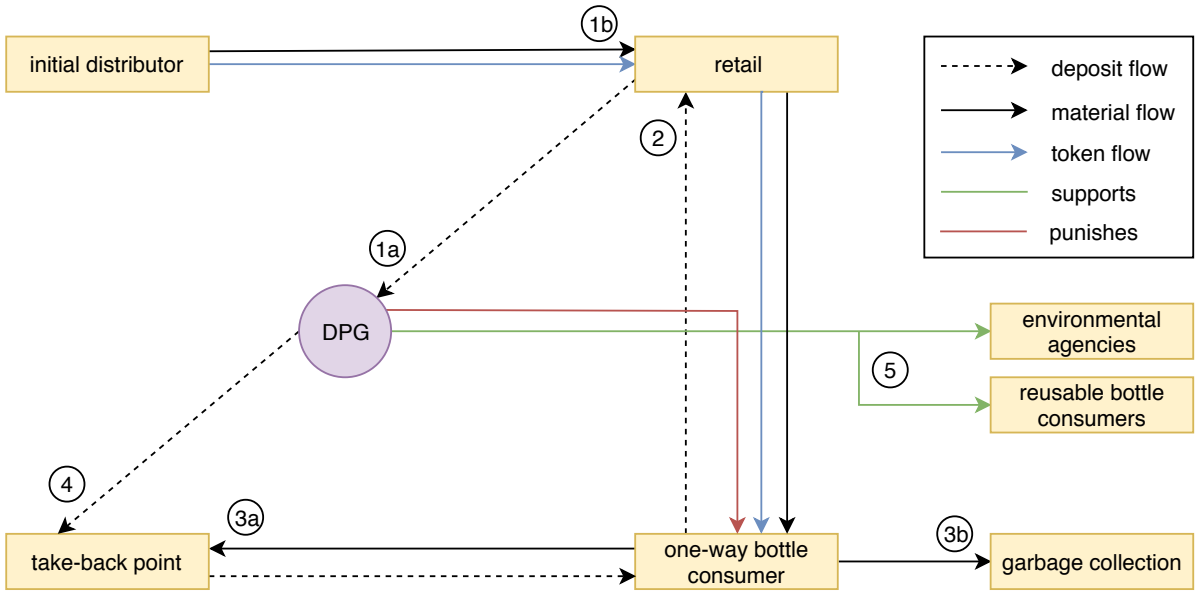

Figure 3.2.: Future deposit-refund cycle [23]

Leveraging Ethereum's decentralized application platform and built-in cryptocurrency (see 2.3.1), DPG could provide participants of the deposit-refund cycle with a fully autonomous method for settling refund claims by deploying a smart-contract that will act as a central clearing house and simultaneously allow for the incentivized structure put forward. A semantic change to the regulatory underpinnings will not be required.

[23] An extended version detailing the three differently balanced accounting states (comp. 2.1.3) can be found in the Appendix B on page 71.

For all this to work, the very first deposit charged for a one-way bottle must be converted to Ether and sent to the smart-contract (step 1a of Figure 3.2). At the same time, a non-fungible token (see 2.2.3) representing the newly introduced bottle must be created using the bottle's unique identifier and associated (step 1b of Figure 3.2) with the address of the purchaser's EOA (see 2.3.3). This means that the only permitted means of identification will be an Ethereum address. Keeping in mind the actor's various obligations (see 3.1.4), the remainder of the system will be implemented as following:

(2) Like today, the deposit of a one-way bottle is still directly paid to the retailer upon each subsequent sale but as already indicated, not again transmitted to the smart-contract. However, should the consumer qualify for a penalty (see 3.1.2), this penalty (converted to Ether) is sent to the smart-contract along with the purchase report itself. Furthermore, each report causes the token to be re-associated with the new consumer. In case that the previous owner had to pay a penalty [24], that amount will be credited to him for later withdrawal as he is not responsible for the bottle anymore.

(3a) By returning a one-way bottle, the accompanying report triggers the smart-contract to compare the consumer's identity with that recorded for the bottle (i.e. that associated with the token). Should these match, the penalty will be credited for later withdrawal. Otherwise, the penalty is withheld (comp. 3.1.2). Irrespective of the outcome, the token is destroyed meaning that the bottle's identifier could be reused in the future. Additionally, tack-back points must physically refund the consumer (comp. Obligation to levy deposits on beverage packaging on page 3)

(3b) Should the consumer decide to throw away a one-way bottle, this information is (likely) to be reported by garbage collection sooner or later and triggers an increase in the responsible consumer's record of thrown away bottles which is used to calculate his penalty (comp. Equation 3.2). Assuming that a penalty had been paid for this bottle, it is withheld (comp. 3.1.2). Lastly, the token is burned and the number of bottles which have been thrown away in the current period is incremented, as is required to correctly calculate Equation 3.1.

(4) Since tack-back points still directly refund consumers, settlement for these advance payments is needed. This happens by transmitting the same signed dataset which is used today and produced either by a reverse vending machine or a commissioned counting center (see 2.1.3). The smart-contract can then verify the payload's signature, after which a message with equivalent value in Ether will be sent to the refund claimant's (i.e. the caller's) address.

[24] Even though unlikely, this indicates that a retailer has lost or thrown away one-way bottles.

(5) Environmental agencies can simply claim their donation by signing a transaction and addressing it to this smart-contract, though, for it to work, the encapsulated message must contain the appropriate function signature (see 2.3.3). Determining that a caller is indeed an environmental agency happens by looking up his approval status. Likewise, eligible consumers (i.e. those that have been reported to have purchased reusable bottles) can claim their reward in the same fashion with the difference being that permission does not have to be checked since the reward amount is not statically calculated (comp. Equation 3.1) and by such would simply result in a zero value.

3.2.2. Component Design

Striving for a component-based design represents one of the most important best practices in software engineering because it facilitates developers in maintaining a complex system by decomposing it into parts that are easier to conceive, understand and program. At the same time, the process of dismantling a system should not happen arbitrarily but rather attempt to take the application's domain structure into account to reduce the likelihood of having to remedy large parts afterwards. Accordingly, Figure 3.3 showcases the individual components that have been identified as being essential to realizing the high-level architecture presented beforehand.

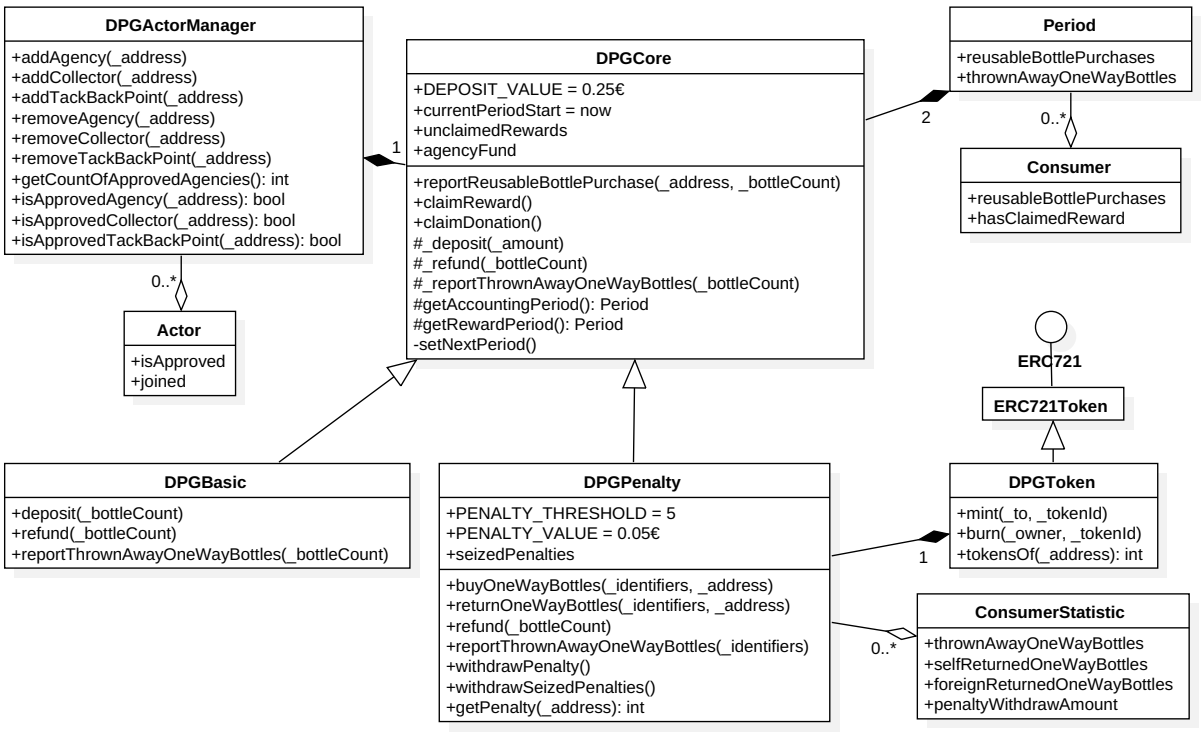

Figure 3.3.: Class design model

The classes (i.e. contracts [58, p. 75]) encountered within this (loosely typed) conceptual model can be categorized by the role they take in abstracting the system:

Accounting

As mentioned in subsection 3.1.1, some kind of an accounting layer will be needed to reward reusable bottle consumers. Here, a smart-contract called `DPGCore` will take upon this task and provide actors with FR-01, FR-03 and FR-04. Since all reported figures are tied to a specific timeframe (i.e. the current period) and rewards can only be claimed for the past timeframe (comp. 3.1.2), `DPGCore` maintains a reference to two distinct `Periods` which each track the total number of bottles that have been thrown away in that specific period, as well as the number of reusable bottles a particular `Consumer` has purchased. The connection to both properties allows FR-08 and FR-09 to be implemented within the same contract. Finally, it makes available two additional functions that for one, enable the settlement process outlined above and two, offer the ability to lock down a deposit that has been transmitted by an initial distributor (comp. 3.2.1).

By taking a closer look at the conceptual model presented, one can make out the fact that it will not be possible to directly put in a claim to be reimbursed, report a thrown away bottle or lock up a deposit because the methods are marked as protected. Consequently, they can only be exposed through inheritance. This behavior is intended considering that the penalty can be regarded as an extension to an incentivized deposit-refund system [25] and by such, requires the APIs to be extended with additional arguments that capture the individual bottle identifiers. Further, this design choice will break up the monolith and simplify deployment should it be decided that a penalty is not needed for production, in which case the subclass `DPGBasic` can be used out of the box.

Access Control

To relieve `DPGCore` of the duty to additionally have to track which addresses resemble approved environmental agencies, garbage collectors or tack-back points, `DPGActorManager` has been established as a separate contract. It is advised that the same or some other key pair in possession of DPG is used to deploy this smart-contract, as only the owner (i.e. the deployer) will be able to add or remove additional `Actors`. Communication between `DPGCore` and `DPGActorManager` will happen by exchanging messages (see 2.3.3). Together, this will ensure NFR-01 in regard to unauthorized donation claims, thrown away bottle reports and bottle returns. Despite these measures, another, more concerning authorization scheme is still undefined.

[25] As the name implies, incentivization primarily refers to rewarding reusable bottle purchases.

Prevention of unauthorized (reusable [26]) bottle purchases could be done in two ways:

- signed purchase receipt (similar to refund claim)
- list of approved retailer addresses

Both options would either require a change to existing cashier systems or mandate retailers on all trade levels to register with DPG in order to capture their identity, as currently, this can only be done for initial distributors and tack-back points (comp. 2.1.3). On the other hand, option number one will solve two additional problems. First, anybody will be able to verify the report's contents and secondly, it can be ensured that the same purchase will not be be reported twice. It is therefore highly recommended that signed receipts are utilized throughout the system, in which case `DPGActorManager` would only have to maintain a list of approved environmental agencies, provided that no unauthorized party can generate such a receipt.

Penalty

Partially discussed above and in subsection 3.2.1, a penalty arises from repeatedly detecting that a consumer has thrown away one-way bottles. Detection occurs after garbage collection reports the identifier of a particular bottle, for which then the corresponding token is retrieved along with its owner. This `DPGToken` will be based on OpenZeppelin's reference implementation of the `ERC721` non-fungible token standard and even though reuse leads to additional project dependencies, it is still very much desirable because the underlying codebase is regularly tested and community-audited [27] [13].

By default, the `ERC721Token` contract does not allow users to mint new or burn existing tokens []. Therefore, subclassing must be used to expose these protected methods. At the same time, a barrier will be needed to prevent arbitrary usage. Generally, it can be argued that tokens shall only be managed by the application as a result of having received a report. To guarantee this behavior, all function calls within `DPGToken` will be restricted to `DPGPenalty`, hereby creating a fully autonomous debt-tracking entity, meaning that no human will be able to transfer their tokens in an attempt to get rid of the return responsibility.

One consequence of separating the penalty's logic from `DPGCore` (comp. Accounting on page 34) is the need for an additional data structure that tracks how many one-way bottles a particular consumer has thrown away. This structure is given by `ConsumerStatistic`. Only now, `DPGPenalty` will be able to implement FR-02, FR-05, FR-06 and FR-07, during which some of the functionality inherited from `DPGCore`

[26] Non-existent authorization would allow attackers to bluntly increase their share in rewards.
[27] $4.5 Billion worth of digital assets are powered by OpenZeppelin smart-contracts [12].

will be re-encapsulated to add the bottle identifier as a function argument. In line with NFR-03, it will be possible to specify multiple bottles at once. Transaction cost (see 2.3.3) reductions like these should be utilized whenever possible.

3.2.3. Conversion of Deposits

To this point, it has been assumed that each deposit will be exchanged for Ether of equal value. Considering Ethereum's volatile nature and that of almost any cryptocurrency, this procedure poses a serious concern when thinking about the reward or settlement process. In the worst case, the fluctuation could diminish all of the locked down funds, requiring DPG to either reject reimbursement claims or go into debt themselves. Two solutions can be thought of depending on the type of blockchain used in production:

Public

When using Ethereum's public permissionless blockchain, fluctuations are inevitable. However, projects known as stablecoins (e.g. Tether, MakerDAO, Digix) attempt to create a currency of constant value. Many of these are backed by a collateral, pegged to traditional fiat currencies or assets (e.g. 1 gram of gold) and most importantly, traded as fungible (see 2.2.3) Ethereum tokens []. Despite each project having its own drawbacks, such a stablecoin could be utilized to better safeguard the funds by:

- acquiring adequate amount of stablecoin tokens upon receiving a deposit
- reimbursing refund claimant with stablecoin tokens

Of course, this would happen completely automatically and could be achieved through a separate proxy contract that interacts with a decentralized exchange before returning control to the actual contract method called.

Permissioned

A more predictable approach involves permissioned blockchains, commonly referred to as private deployments. These are operated by an authorized set of miners who use Ether at constant value []. Ergo, the solution can be used as is, though assumes that:

- DPG operates a public fixed-price exchange to buy/sell internal Ether
- actors and consumers can freely transact and create accounts

The system would then be used with the following peculiarities:

- initial distributors exchange deposit for adequate amount of internal Ether

- agencies, consumers and claimants exchange their internal Ether when needed

Now, one may suggest that a similar approach could be taken in a public blockchain environment, if a second token representing a constant deposit value is used. The problem, however, is that tokens cannot be sent along as a regular function argument. This is due to the fact that tokens are merely a recoding in application state which means that a particular contract has used its storage to remember how many tokens any given address possesses []. At the same time, this is also completely true for Ether itself with the exception being that miners automatically deduct one's balance should they encounter that a transaction has specified a non-zero amount of Ether [].

3.2.4. Design Rationale

Accountless

Modeling the entire system as a series of interactions with a smart-contract is a deliberate design choice because, by definition, it prescribes all users to be authenticated with each request. This is due to the fact that only the private-key owning entity can sign transactions for that particular address (comp. 2.3.3) which allows for effortless authorization simply by maintaining a list of approved addresses. More importantly, consumers will not be required to register with yet another service just to leave their bank account details on a centralized server that is definitely more vulnerable to attacks than the decentralized ownership model embodied by public-key cryptography [28].

Public Blockchain Value-Add

From here, the general advantages of deploying the solution on a public blockchain should be highlighted with respect to the non-functional requirements defined in subsection 3.1.5.

Availability (NFR-02)

The application will most certainly experience 100% availability because the likelihood of all 17.6k miners (comp. 2.3.2) going offline is extremely low. As such, it will not be necessary that clients maintain a backlog of unsent reports. Furthermore, the transaction pool itself will act as a backlog, should Ethereum experience an overly high demand in which it is unable to directly process a request.

[28] A secure, non-constantly connected storage medium offers the greatest degree of protection against private-key theft. Devices, known as hardware wallets (e.g. Ledger Nano S, Trezor), have been specifically designed for this purpose [].

Cost (NFR-03)

Assuming a constant gas price and distribution of request types, the operating costs will always be proportional to the application's traffic (i.e. number of contract invocations) as only the actual computation expended by a miner will be billed (comp. 2.3.3). In this sense, the one-time fee paid for deployment is negligible [].

Efficiency (NFR-05)

Presuming that a miner will not randomly leave out transactions and always prefer transactions with a higher gas price, any request will execute by the end of the day, if its gas price has been set high enough.

Integrity (NFR-06)

Provided that the contract code is error-free, any report will be recorded as is because all transactions are atomic []. Further, all input arguments are strongly typed [], so that, if required, only integers will be accepted.

Withdrawal Pattern

There are two particular reasons as to why refunded penalties are credited for later withdrawal instead of being sent to the owner directly or paid out as part of the bottle return process. First, the withdrawal pattern generally represents an Ethereum best practice because it avoids reentrancy attacks []. This is also one of the reasons why rewards and donations have to be withdrawn. Secondly, it would be very impractical to incorporate this step into the bottle return process as reverse vending machines would otherwise have to look up the penalty and convert it to fiat. Alternatively, the amount could be refunded directly, although this would require tack-back points to keep an additional record and submit that record as part of a refund claim, necessitating even more changes to the current infrastructure.

4. Implementation

4.1. Limitations

The following chapter will attempt to highlight the most important implementation details and explain which considerations have lead to the solution given in [14] [29]. This product should by no means be regarded as final since two important aspects are uncovered:

- conversion of deposits (see 3.2.3)
- signed receipts (see Access Control on page 34)

Also, it must be noted that all programming has been done in Solidity, a statically typed, high-level language that was influenced by C++, Python and JavaScript [58, p. 5]. Alternatives have not been considered, because Solidity represents Ethereum's most popular language [25, p. 21].

[29] The accompanying class implementation model can be found in the Appendix C on page 73.

4.2. Accounting

realized via `DPGCore`

In this context, accounting refers to the task of tracking how many reusable bottles a particular consumer has bought in a given timeframe, while also noting down the total number of one-way bottles that have been thrown away in that same timeframe (comp. Accounting on page 34).

4.2.1. A/B Scheme

Subsection 3.1.2 states that the timeframe should be set to 4 weeks which has literally been done so by utilizing one of the six time unit suffixes available [58, p. 68]:

```
1 uint public constant PERIOD_LENGTH = 4 weeks;
```

Listing 4.1: Definition of period length

As previously explained, it will only be necessary to keep an account of the two most recent periods (comp. Accounting on page 34). By such, the application can be thought of as alternating between two data structures of which one represents the current period used for incoming reports, while the other is stored to determine a consumer's reward. Of course, this will require a reset of the older period as soon as a switch takes place. Figure 4.1 aims to illustrate this behavior as time progresses.

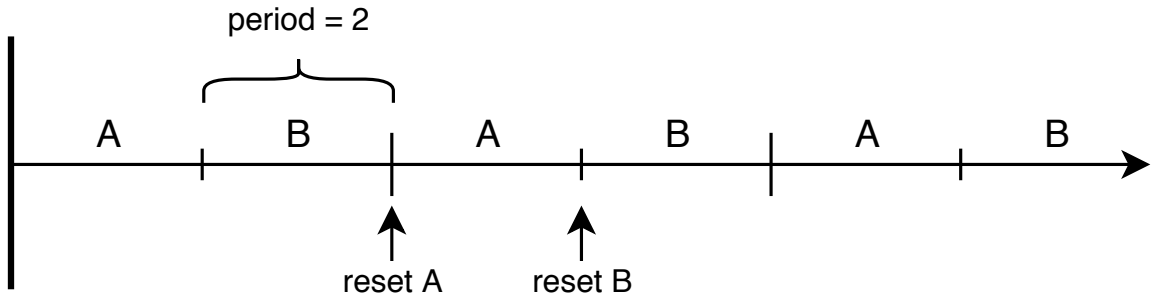

Figure 4.1.: Accounting timeline

Internally, switching is realized by using a third variable called `currentPeriodName` that tracks which of the two data structures (i.e. A or B) should be used for new reports. For clarity, an enumeration called `PeriodName` is defined with the same possibilities. Accordingly, the opposite of the current value will then represent the past period. This technique is detailed in Listing 4.2.

```
enum PeriodName { A, B}

PeriodName private currentPeriodName = PeriodName.A;
Period private periodA;
Period private periodB;
```

Listing 4.2: Declaration of periods and definition of current period tracker

In order to avoid repetition, two additional helper functions are implemented to provide other methods with a constant access interface. Listing 4.3 showcases how this is done in respect to the current period. Again, the inverse logic will yield the past period.

```
function getAccountingPeriod() internal view returns (Period storage) {
  return currentPeriod == PeriodName.A ? periodA : periodB;
}
```

Listing 4.3: Helper function to retrieve current period

Here, the function's return type is marked as `storage`, meaning that it will return a reference to the period (stored in the contract's persistent storage [see 2.3.3]), not a copy of it. This greatly simplifies persistence as changes do not have to be written back manually. Alternatively, one could also directly maintain two storage pointers that reference the data structures in a logical manner. The advantage of that approach is that no helper function will be required for retrieval, though it might obscure how the application works and therefore, contradicts with a dApp's goal of establishing trust (comp. 2.2.2). Still, both possibilities have been confirmed to work equally.

At this point, it should be noted that a function declared as `view` promises to not modify the contract's state [30] [58, p. 82]. This commitment can be clearly seen in Listing 4.3 but is also enforced by the EVM itself [58, p. 83]. Practically, this allows nodes to perform synchronous read-only operations that are free of charge [15]. On the contrary, calling such a function by means of transactions will still cost gas because the transaction must be stored in the blockchain even if it does not modify that particular contract storage [16].

4.2.2. Period Reset

Traditionally, resetting a period would be achieved by assigning it a new instance. Sadly, this method is not feasible because the `Period` data structure uses a mapping to assign each (consumer) address a `Consumer` data structure as can be seen in Listing 4.4.

[30] `Pure` functions additionally disallow access of state [58, p. 82].

```
struct Period {
  uint index;
  uint reusableBottlePurchases;
  uint thrownAwayOneWayBottles;
  mapping(address => Consumer) consumers;
}
```

Listing 4.4: Period data structure

Like often, this limitation can be traced back to gas (see 2.3.3). Considering that mappings are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros [58, p. 62], this behavior is explained by the fact that one cannot provide enough gas to pay for the 2^{256} operations needed for a reset without exceeding the block's `GASLIMIT` (see footnote 21). Therefore, `delete` has no effect on whole mappings [58, p. 63].

```
struct Consumer {
  uint reusableBottlesPurchases;
  bool hasClaimedReward;
  uint lastResetPeriodIndex;
}
```

Listing 4.5: Consumer data structure

A simple workaround is given by Listing 4.5 in which an additional property called `lastResetPeriodIndex` is associated with each consumer to remember the period (index) in which the consumer's attributes were last reset. Should this index be smaller than the contract's `currentPeriodIndex`, a reset must occur before any attribute may be updated. Logically, the `lastResetPeriodIndex` must then be set to the `currentPeriodIndex` to prevent the consumer from being reset each time.

4.2.3. Period Advancement

Solidity function modifiers can be used to automatically check a condition prior to executing the function [58, p. 79]. This application implements a modifier called `periodDependent` to ensure that reports always attribute to the most current period and do not change the results of the past one. By such, it may be regarded as the switch in the A/B scheme outlined before.

Practically, this means that the application will check whether the current period's duration has exceeded the `PERIOD_LENGTH` (see Listing 4.1) and if so, switches periods before

returning control to the method called which is symbolized by an underscore [58, p. 81]. Listing 4.6 displays this logic, where `now` is an alias for the current block timestamp [31], `setNextPeriod()` the function used to actually switch periods and `currentPeriodStart` self-explanatory. All timestamps are given in unix epoch format [58, p. 29].

```
modifier periodDependent() {
  if (now < currentPeriodStart.add(PERIOD_LENGTH)) {
    _;
  } else {
    setNextPeriod();
    _;
  }
}
```

Listing 4.6: Function modifier to advance period

Finally, Listing 4.7 demonstrates how this modifier is applied to functions.

```
function reportReusableBottlePurchase(address _address, uint
    _bottleCount) public periodDependent
```

Listing 4.7: Function signature to report reusable bottle purchase

[31] Timestamps can be influenced by miners and are only guaranteed to be somewhere between the timestamps of two consecutive blocks of the canonical chain [58, p. 65].

4.3. Access Control

realized via `DPGActorManager`

In its most basic form, access control can be realized by maintaining a list of authorized addresses. For an incentivized deposit-refund system, this list must at least cover all environmental agencies who should be able to claim a donation (comp. Access Control on page 34).

4.3.1. Storage Layout

Even though access control could be realized through a simple list (i.e. an array), gas consumption diminishes this approach's efficiency because the average cost of checking whether a given address is contained in the list is linear to the number of approved addresses (i.e. `O(n)`). Moreover, this cost cannot be predicted and may very well exceed the block's `GASLIMIT` causing the contract to be stalled [58, p. 122]. Instead, a mapping is used to ensure `O(1)` lookups.

```
1 struct Actor {
2   bool isApproved;
3   uint joined;
4 }
5
6 mapping(address => Actor) private collectors;
```

Listing 4.8: Declaration of approved agencies and Actor data structure

4.3.2. Ownable

As mentioned in subsection 3.2.2, only DPG should be able to add or remove approved parties. Again, this is easily realized through modifiers and even promoted as one of its premiere use cases [58, p. 80]. Because this functionality will be needed quite often (comp. Figure C.1 on page 73), the project extracts the logic into a separate contract that can be inherited from. From Listing 4.9 it can be seen that only the address passed as part of the constructor will be able to call a function that is marked as `onlyOwner`. In all other cases, the call will revert and refund the remaining gas since the `require` statement cannot be fulfilled [58, p. 75].

```
contract Ownable {
  address public owner;

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  constructor(address _owner) public {
    owner = _owner;
  }
}
```

Listing 4.9: Ownable contract

Specifying which address should become the contract owner may happen either directly in the inheritance list or as part of the derived constructor, known as the modifier-invocation-style. This is true for all constant base constructor arguments [58, p. 89]. Both options are shown in Listing 4.10, where `msg.sender` is the sender of the message (or transaction which encapsulated the message [comp. 2.3.3]). Consequently, both lead to the same desired result in which the deployer becomes the owner.

```
// inheritance list
contract DPGActorManager is Ownable(msg.sender) { }

// modifier-invocation-style
contract DPGActorManager is Ownable {
  constructor() public Ownable(msg.sender) {}
}
```

Listing 4.10: Inheriting from Ownable contract

4.3.3. External Interface

One result of extracting functionality into a separate contract is the need for an external interface that specifies which functions can be called by other contracts. For this purpose, Solidity provides two types of visibilities: `external` and `public`, the difference being that `external` functions cannot be called internally [58, p. 77]. What this implies will be explained in the following [58, pp. 57, 69]:

Internal Call (`f()`)

Internal function calls are translated into EVM jump instructions. The current

memory is not cleared and by such, allows references to be passed very efficiently. Only functions within the same contract can be called like this.

External Call (`this.f()`)

External function calls are executed via a message call. All function arguments have to be copied to the `calldata` location which is a non-modifiable, non-persistent area used solely for function arguments. Callers can specify the amount of Wei and gas.

A general recommendation is to use `external` if that function is expected to only ever be called externally, meaning that it will not be needed by the contract itself [17]. Moreover, `external` functions are sometimes more efficient when dealing with large arrays of data [58, p. 77]. Last but not least, `external` functions have the potential to reduce upfront storage costs. Consider the following example:

```
1  import "./DPGActorManager.sol";
2
3  contract DPGCore {
4    DPGActorManager internal actorManager;
5
6    constructor(address _actorManager) public {
7      actorManager = DPGActorManager(_actorManager);
8      ...
9    }
10 }
```

Listing 4.11: Sharing external contract interface via inheritance

From Listing 4.11 it can be seen that `DPGCore` declares a variable called `actorManager` which is of type `DPGActorManager`. This variable will be defined during construction by taking the address of an already deployed `DPGActorManager` instance and casting it to this type. Behind the scenes, `DPGCore` will be forced to inherit all of the bytecode from `DPGActorManager`, resulting in a potentially large code its own [18], even though none of the `private` or `internal` properties of `DPGActorManager` can be used.

On the other hand, interfaces will limit this inheritance to the critical Application Binary Interface (ABI) which is the standard way to interact with contracts, both from outside the blockchain and for contract-to-contract interaction [58, pp. 93, 133]. The bytecode will be much smaller and less likely to run into an out of gas exception when being deployed [18]. Listing 4.12 demonstrates this alternative approach.

```
interface IDPGActorManager {
  function isApproved(address _address) external view returns (bool);
  ...
}

contract DPGActorManager is IDPGActorManager { ... }

contract DPGCore {
  IDPGActorManager internal actorManager;

  constructor(address _actorManager) public {
    actorManager = IDPGActorManager(_actorManager);
    ...
  }
}
```

Listing 4.12: Sharing external contract interface via interfaces

However, it must be noted that interfaces are limited to the aforementioned `external` functions and cannot define variables, structs or enums [58, p. 92]. Still, this method of sharing a contract's interface will be preferred as it supports NFR-03.

4.4. Penalty

realized via `DPGPenalty & DPGToken`

A consumer must face a penalty if the system repeatedly detects that he has thrown away his one-way bottles. Detection occurs after garbage collection reports the identifier of a bottle. Tracking who is responsible for a particular bottle happens by means of a non-fungible token which is associated with the consumer's identity upon purchase (comp. Penalty on page 35).

4.4.1. ERC-721 Token Standard

Non-fungible tokens are characterized by the fact that they cannot be substituted without incurring a difference in value, function or meaning. They represent unique tangible or intangible items (comp. 2.2.3). In this case, such a token will represent a unique bottle that is tangible indeed. At the same time, this means that the very popular ERC-20 standard cannot be used because it was designed for fungibility [19].

Before an alternative standard is presented, it should be mentioned that all token standards are a minimum specification for an implementation. They do not prescribe an exact implementation and do not prevent developers from adding functionality on top. Token standards encourage interoperability by agreeing on a set of functional requirements [25, pp. 198–199].

ERC-721 is a proposal to standardize non-fungible tokens, also known as deeds. It places no limitation or expectation on the nature of the thing whose ownership should be tracked, only that it can be uniquely identified [25, p. 197]. Internally, this identifier will be a 256-bit unsigned integer which provides more than enough space for growth. The complete contract interface specification is given in [20]. Here, only the most important methods shall be highlighted for discussion:

```
interface ERC721 {
    function balanceOf(address _owner) external view returns (uint256);
    function ownerOf(uint256 _tokenId) external view returns (address);
    function safeTransferFrom(address _from, address _to, uint256
        _tokenId) external payable;
    ...
}
```

Listing 4.13: ERC-721 contract interface specification

Compared to ERC-20, two major differences can be found. First, a new `safeTransferFrom` method is introduced. This method checks if the receiving party is a smart-contract and if so, will call `onERC721Received`, expecting the contract to confirm the transfer by echoing the function signature. Secondly, metadata such as the name and symbol is absent. This metadata, together with a new method that should return an asset's URI, has been moved into a separate optional interface called `ERC721Metadata`. For enumeration purposes, another optional interface has been defined as `ERC721Enumerable`.

4.4.2. OpenZeppelin Framework

Basic standards like ERC-20 and ERC-721 have emerged from experience with hundreds of applications and fit 99% of the use-cases. Choosing a standard like this will not only gain the value of the systems designed to work with it, but also allow developers to reuse and extend battle-tested reference implementations [25, pp. 199–200].

OpenZeppelin provides dApp developers with a library of smart-contracts that are tested and community-reviewed [12]. This project utilizes two of its components:

SafeMath.sol
: Math operations with safety checks that revert on error

ERC721Token.sol
: Implements all the required and some optional functionality of the ERC721 standard

Installation via NPM is simple and quick, though it is recommended that the `-save-exact` flag is used to avoid breaking changes because the library does not follow semantic versioning.

4.4.3. Token Transfer Restriction

Having settled on a token standard and reference implementation, this section details the adjustments that have been made to guarantee the application's promise of establishing a fully autonomous debt-tracking entity (comp. Penalty on page 35).

By default, ERC-721 only permits the owner of a token to initiate a transfer. Owners can also authorize an address to transfer a specific token on their behalf or even appoint an operator who is free to decide about all of the owner's tokens [20]. OpenZeppelin's implementation abstracts these authorization rules with a single method named `isApprovedOrOwner(address, uint256)`, where `uint256` refers to the token and `address` represents the caller who wishes to transfer it [13].

In this application context, none of the options should be accessible since they would allow refusal of ownership. To prevent this from happening, two changes have been introduced:

- inherit from `Ownable` and mark all state-modifying methods as `onlyOwner`
- return `true` in `isApprovedOrOwner(address, uint256)`

This approach infers that Solidity supports multiple inheritance since `DPGToken` must now inherit from both `ERC721Token` and `Ownable`. Listing 4.14 demoes how multiple inheritance is declared. Internally, this is implemented by copying code which means that only a single contract will be created on the blockchain [58, p. 88].

```
contract DPGToken is ERC721Token, Ownable {
  constructor() public ERC721Token("DPG Token", "DPG") Ownable(msg.
      sender) {}
  ...
}
```

Listing 4.14: Multiple inheritance by DPGToken

Finally, it will be necessary that `DPGPenalty` deploys the `DPGToken` contract on its own. Otherwise, it will not be set as the owner (comp. 4.3.2). Creating a new contract on the blockchain is done via the `new` keyword and requires the full code to be known in advance [58, p. 71]. Optimizations like in subsection 4.3.3 are not possible.

```
contract DPGPenalty is DPGCore {
  DPGToken public token = new DPGToken();
  ...
}
```

Listing 4.15: Creating a contract from a contract

5. Testing

5.1. Test-Driven Development

Test-driven development hopes to help programmers clarify what a code segment is actually supposed to do by requiring the accompanying tests to be written beforehand. Moreover, code is developed incrementally, meaning that the next identified functionality is not implemented until the previous code passes all of its tests. Combined with an automated testing environment, this leads to the following benefits:

- Tests showcase which segments have actually been executed (*coverage*)
- Tests can be rerun to ensure that changes have not introduced new bugs (*regression*)
- Tests pinpoint the root causes of problems (*debugging*)
- Tests describe what the code should do (*documentation*)

Given the scale of the project, it has been decided to mainly rely on requirements-based testing which is a semantic approach to test-case design where a set of tests is derived for each requirement. In a second step, scenario testing has been employed to develop test cases that cover realistic scenarios of use to which most people should be able to relate. By such, the testing methodology can be categorized as black-box testing. Like always, testing can only show the presence of errors, not their absence.

Suite Name	Components	# of Tests	Functional Requirements
Deposit Refund	Core, Actor	7	-
Report Garbage	Core, Actor	12	FR-01, FR-02
Report Purchase	Core, Actor	19	FR-03, FR-04
Penalty	Penalty, Actor, Token	29	FR-05, FR-06, FR-07
Claim Reward	Core, Actor	7	FR-08
Claim Donation	Core, Actor	11	FR-09

Table 5.1.: Test suites

In total, 85 test cases have been devised, all of which are available at [14]. Table 5.1 breaks down this number onto the individual test suits, components and functional requirements.

5.2. Truffle Suite

The Truffle Suite plays an important role in the test-driven development process which will be demoed in the following section. First, two of its tools must be introduced:

Truffle

Truffle is a development environment, testing framework and asset pipeline for blockchains using the EVM [21, Truffle Overview].

Ganache

Formerly known as TestRPC, Ganache simulates an Ethereum blockchain that, by default, mines transactions as soon as they are received [21, Ganache Quickstart].

5.2.1. Network Management & Deployment

In addition to basic compilation and linking, Truffle provides developers with a one-step process for deploying multiple smart-contracts to both public and private Ethereum networks [21, Truffle Overview]. These networks can be specified within a `truffle.js` file [21, Configuration], as is given below.

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" // match any network ID
    }
  },
};
```

Listing 5.1: Truffle network management

Since deploying to the public Ethereum main-chain involves real fees (comp. 2.3.3), Ganache has been used throughout development. At first sight, one may believe that the `host` specified in Listing 5.1 symbolizes this intention because it specifies a localhost IP address. However, the `host` only tells Truffle where an Ethereum client resides [21, Configuration], not what kind of type it is. Furthermore, Ganache and most other Ethereum clients (e.g. geth and parity) default to the same `port` to expose RPC services. The consequence is, that developers must either stop those clients or set Ganache to a different port in order to prevent accidental deployment [21, Deploying To The Live Network].

Before deployment finally becomes possible, it is necessary to create a migration script that is responsible for staging the deployment tasks [21, Running Migrations]. Listing 5.2 displays the script used in this project.

```
var DPGActorManager = artifacts.require("./DPGActorManager.sol");
var DPGPenalty = artifacts.require("./DPGPenalty.sol");

module.exports = async function(deployer, network, accounts) {
  await deployer.deploy(DPGActorManager);
  await deployer.deploy(DPGPenalty, DPGActorManager.address);
};
```

Listing 5.2: Truffle migration script

As previously explained, `DPGCore` expects it to be passed the address of an already deployed `DPGActorManager` instance (comp. Listing 4.11). By virtue of subclassing, this requirement translates to `DPGPenalty` (comp. Figure 3.3). Therefore, the script `awaits` the deployer to finish that task before actually deploying the main contract. Remember, `DPGPenalty` autonomously deploys the `DPGToken` contract (comp. 4.15). Hence, it is not part of this migration.

From here, one must simply start Ganache, optionally specify its port (`-p`) and use Truffle to deploy all contracts with a single command. This process is shown in Listing 5.3.

```
ganache-cli -p 7545
truffle migrate
```

Listing 5.3: Deploying to Ganache

5.2.2. Automated Contract Testing

Having an automated testing environment is essential for test-driven development. At the same time, this stipulates that tests are written as an automated test, meaning that they will report whether they have passed or not. Other desirable attributes are given by the F.I.R.S.T. rule [45, p. 132]:

Fast

Tests should be executed quickly, so that they can be run more often.

Independent

Tests should be independent of each other, so that they can be run in any order.

Repeatable

Tests should be repeatable in every environment independent of network status.

Self-Validating

Tests should either pass or fail and not require external judgement.

Timely

Tests should be written early to simplify development.

Truffle comes standard with an automated testing framework that allows developers to write simple and manageable tests in either JavaScript or Solidity. It provides a clean-room environment while doing so to ensure that state will not be shared across test suites. Further, all tests running on Ganache will be able to utilize snapshotting which results in an up to 90% speed improvement compared to redeploying contracts. Still, it is recommended to also run tests against an official Ethereum client as release comes closer [21, Testing Your Contracts].

Right in line with the requirements and scenario-based testing outlined in section 5.1, this project has settled on JavaScript tests because they are meant for exercising contracts from the outside world [21, Testing Your Contracts]. At its core, JavaScript tests are based on the popular Mocha testing framework [21, Writing Tests in JavaScript].

Example

To demonstrate Truffle's capabilities in practice, the following scenario shall be tested:

> It should fail to send the reward to consumer A (even though consumer A purchased 3 reusable bottles in the past period) because the reward amount is not greater than 0 as no report of thrown away bottles was received in the past period.

This test will be part of the `DPG Claim Reward Test` suite (see Table 5.1). Suites are identified by the `contract` function which is a special abstraction of Mocha's `describe()` environment needed to provide aforementioned clean-room features. In order to send transactions from multiple EOAs, the `contract` function gets passed the `accounts` of the underlying Ethereum client [21, Writing Tests in JavaScript].

First off, a test suite must import all of its required components. This is done by using the `artifacts.require()` method which requests Truffle to generate a JavaScript interface for the specified Solidity contract [21, Writing Tests in JavaScript]. Both import and setup of a suite are demonstrated in Listing 5.4.

```
const DPGBasic = artifacts.require("DPGBasic");
const DPGActorManager = artifacts.require("DPGActorManager");

contract("DPG Claim Reward Test", async (accounts) => {
  let mainContract;
  let actorManagerContract;

  // hooks
  beforeEach("redeploy contract, setup deposits ...", async() => {
    actorManagerContract = await DPGActorManager.new();
    mainContract = await DPGBasic.new(actorManagerContract.address);

    await mainContract.deposit(10, {from: retail, value: 10});
    await actorManagerContract.addCollector(collectorA, {from: owner});
  });

  ...
}
```

Listing 5.4: Setup of test suite

Notably, Listing 5.4 also defines a so called `beforeEach` hook. As the name implies, this method will be executed before each test. By extracting common setup steps, the test becomes independent as it no longer relies on the setup that a previous test would have otherwise performed. Here, setup involves redeployment of both contracts, putting in a deposit for 10 bottles and adding `collectorA` as an approved garbage collector. The last argument of each contract method may be provided as a JSON so that developers can specify `from` which of the `accounts` the transaction should be sent and what `value` in Wei will be transferred [21, Interacting With Your Contracts].

Now, the actual test may be designed. Its logic is relatively simple and self-explanatory when taking a look at Listing 5.5. Two particular details should be noted however. First, a custom `timeTravel` method has been implemented (outside of this suit) to bidirectionally travel in time by sending an `evm_increaseTime` message to the Ethereum client. A known bug then requires a consequent `evm_mine` message to force the change of system time into effect [52].

```
it("should ...", async() => {
    await mainContract.reportReusables(consumerA, 5, {from: retail});

    await timeTravel(86400 * 28);
    await mineBlock();

    try {
        await mainContract.claimReward({from: consumerA});
    } catch (error) {
        return true;
    }

    throw new Error("Did send the reward even though ...");
});
```

Listing 5.5: Example test case

Obviously, this test is self-validating because it will either fail with an error or succeed by returning true. Assuming that Ganache has already been started, the complete test suite can be run by entering `truffle test` [21, Testing Your Contracts].

5.3. Continuous Testing

Continues testing describes the process of executing automated tests as part of the software delivery pipeline [26]. It should be considered a prerequisite for continuous delivery and integration [33].

Despite the limited nature of this project, a small pipeline has been set up with the help of [50] to showcase the maturity of Ethereum's developer ecosystem and demonstrate that enterprise-grade workflows can be realized. Leveraging GitHub's tight integration with TravisCI, a free continuous integration tool for public repositories [22], the following actions were triggered upon each commit:

- smart-contract compilation
- test suite execution
- code coverage report (available at [23])

This workflow has protected against unwanted regression when developing new features or refactoring code. Moreover, it has given the number of passing tests greater meaningfulness since those may have only covered 10% of the total codebase.

6. Conclusion and Discussion

7. Summary

8. Outlook

8.1. Enhancements and Additions

8.2. Adoption and Scalability

8.3. Fields of Application

Bibliography

[1] 2014. URL: http://www.johnstonslaw.org/.

[2] URL: https://www.duh.de/mehrweg-klimaschutz0/einweg-plastikflaschen/.

[3] URL: https://www.reiling.eu/news.php.

[4] URL: https://github.com/dmehrotra/fuck-off-aws.

[5] URL: https://web3.foundation/.

[6] URL: https://stackexchange.com/sites#technology-traffic.

[7] URL: https://bp.eosgo.io/explore/?search_keywords=&job_category%5B%5D=block-producer&tab=search-form&type=place.

[8] URL: https://neo.org/consensus.

[9] URL: https://www.ethernodes.org/network/1.

[10] URL: https://github.com/s-tikhomirov/smart-contract-languages.

[11] URL: https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[12] URL: https://openzeppelin.org/.

[13] URL: https://github.com/OpenZeppelin/openzeppelin-solidity.

[14] URL: https://github.com/niksauer/DepositRefund.

[15] URL: https://ethereum.stackexchange.com/questions/765/what-is-the-difference-between-a-transaction-and-a-call.

[16] URL: https://ethereum.stackexchange.com/questions/13851/could-we-call-a-constant-function-without-spending-any-gas-inside-a-transaction/13855.

[17] URL: https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices.

[18] URL: https://ethereum.stackexchange.com/questions/26674/deploying-abstract-contracts-and-interfaces.

[19] URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[20] URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md.

[21] URL: https://truffleframework.com.

[22] URL: https://github.com/marketplace/travis-ci.

[23] URL: https://coveralls.io/github/niksauer/DepositRefund?branch=master.

[24] Jake Adelstein. *Japan's Financial Regulator Is Pushing Crypto Exchanges To Drop 'Altcoins' Favored By Criminals*. Forbes. Apr. 2018. URL: https://www.forbes.com/sites/adelsteinjake/2018/04/30/japans-financial-regulator-is-pushing-crypto-exchanges-to-drop-altcoins-favored-by-criminals/.

[25] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. Ed. by Mike Loukides. First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., Apr. 2018. ISBN: 9781491971949.

[26] Adam Auerbach. *Part of the Pipeline: Why Continuous Testing Is Essential*. Aug. 2015. URL: https://www.techwell.com/techwell-insights/2015/08/part-pipeline-why-continuous-testing-essential.

[27] James Ball, Julian Borger, and Glenn Greenwald. *Revealed: how US and UK spy agencies defeat internet privacy and security*. The Guardian. Sept. 2013. URL: https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security.

[28] Russel Brandom. *Using the internet without the Amazon Cloud*. The Verge. July 2018. URL: https://www.theverge.com/2018/7/28/17622792/plugin-use-the-internet-without-the-amazon-cloud.

[29] Bundesministerium für Umwelt, Naturschutz und Reaktorsicherheit. *Mehrweganteile am Getränkeverbrauch nach Getränkebereichen in den Jahren 1991 bis 2013 (in %) in der Bundesrepublik Deutschland*. 2015. URL: https://www.bmu.de/fileadmin/Daten_BMU/Bilder_Infografiken/verpackungen_mehrweganteile_bf.pdf.

[30] *Bundesweite Erhebung von Daten zum Verbrauch von Getränken in Mehrweg- und ökologisch vorteilhaften Einweggetränkeverpackungen für die Jahre 2014 und 2015*. Texte 52/2017. Dessau-Roßlau: Umweltbundesamt, Feb. 2017.

[31] Vitalik Buterin. *DAOs, DACs, DAs and More: An Incomplete Terminology Guide*. Ethereum Foundation. May 2014. URL: https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide/.

[32] Vitalik Buterin. *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform*. ethereum.org, 2014.

[33] Christina Cardoza. *Continuous testing in a fast-paced agile and DevOps world*. SDTimes. Apr. 2017. URL: https://sdtimes.com/agile/continuous-testing-fast-paced-agile-devops-world/#sthash.TkBEHNKA.dpuf.

[34] Julia Chatterley, Scarlet Fu, and Joseph Lubin. *Ethereum Co-Founder on Bitcoin and Blockchain Tech*. Bloomberg. Dec. 2017. URL: https://www.bloomberg.com/news/videos/2017-12-15/ethereum-co-founder-on-bitcoin-and-blockchain-tech-video.

[35] *Dosenpfand: Handel will die Wahrheit nicht wissen*. SPIEGEL ONLINE GmbH. Sept. 2001. URL: http://www.spiegel.de/politik/deutschland/dosenpfand-handel-will-die-wahrheit-nicht-wissen-a-156398.html.

[36] *DPG Deutsche Pfandsystem GmbH*. Wikipedia.

[37] *Dritte Verordnung zur Änderung der Verpackungsverordnung*. Bundesgesetzblatt Teil I 29/2005. Bundesanzeiger Verlag, May 2005.

[38] Fritz Flanderka, Haucke Schlüter, and Joachim Quoden. *Verpackungsverordnung: Kommentar*. Praxis Umweltrecht 8. Heidelberg: C.F. Müller Verlag, 1999.

[39] Fritz Flanderka, Clemens Stroetmann, and Frank Sieberger. *Verpackungsverordnung: Kommentar für die Praxis unter vollständiger Berücksichtigung der 5. Änderungsverordnung*. 3. Auflage. Heidelberg: C.F. Müller Verlag, 2009.

[40] *Fünfte Verordnung zur Änderung der Verpackungsverordnung*. Bundesgesetzblatt Teil I 12/2008. Bundesanzeiger Verlag, Apr. 2008.

[41] Ian Grigg. *EOS - An Introduction*. block.one. July 2017.

[42] Uta Hartlep and Rainer Souren. *Recycling von Einweggetränkeverpackungen in Deutschland: Gesetzliche Regelungen und Funktionsweise des implementierten Pfandsystems*. Ilmenauer Schriften zur Betriebswirtschaftslehre 2/2011. Ilmenau: Verl. proWiWi, 2011. ISBN: 978-3-940882-27-1.

[43] David Johnston et al. *The General Theory of Decentralized Applications, Dapps*. Feb. 2015. URL: https://github.com/DavidJohnstonCEO/DecentralizedApplications.

[44] Yutu Liu, Anne H Ngu, and Liang Z Zeng. "QoS computation and policing in dynamic web service selection". In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters - WWW Alt. '04*. WWW Alt. '04. New York, New York, USA: ACM Press, 2004, p. 66. ISBN: 1581139128.

[45] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. First Edition. Prentice Hall, Aug. 2008.

[46] A. L. Martínez-Ortiz et al. "A quality model for web components". In: *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services - iiWAS '16*. iiWAS '16. New York, NY, USA: ACM, 2016, pp. 430–432. ISBN: 9781450348072.

[47] Everett Muzzy and Jeff Gillis. *The State of the Ethereum Network*. ConsenSys. June 2018. URL: https://media.consensys.net/the-state-of-the-ethereum-network-949332cb6895.

[48] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Bitcoin.org, Oct. 2008.

[49] *NEO Entered the Era of Decentralization*. neo.org. July 2018. URL: https://neo.org/blog/details/4089.

[50] Gareth Oates. *Testing and Code Coverage of Solidity Smart Contracts*. Aug. 2018. URL: https://medium.com/edgefund/testing-and-code-coverage-of-solidity-smart-contracts-660cb6291701.

[51] Albrecht Patrick et al. *Mehrweg- und Recyclingsystem für ausgewählte Getränkeverpackungen aus Nachhaltigkeitssicht*. PricewaterhouseCoopers AG WPG. June 2011.

[52] Angello Pozo. *Testing Solidity with Truffle and Async/Await*. Aug. 2017. URL: https://medium.com/coinmonks/testing-solidity-with-truffle-and-async-await-396e81c54f93.

[53] Siraj Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. Ed. by Tim McGovern. First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., Aug. 2016. ISBN: 9781491924549.

[54] T. Rummler and W. Schutt. *Verpackungsverordnung: Praxishandbuch mit Kommentar*. Hamburg, 1991.

[55] Niklas Sauer. *File Auditing: A Blockchain Based Approach*. Sept. 2016.

[56] Larry Seltzer. *Serverless computing explained*. Hewlett Packard Enterprise. July 2018. URL: https://www.hpe.com/us/en/insights/articles/serverless-computing-explained-1807.html?jumpid=em_khs4py1ic5_aid-510366305#.

[57] Alexander Smoltczyk and Matthias Geyer. "Die Dosenrepublik". In: *Der Spiegel*. 32/2003. Aug. 2003.

[58] *Solidity Documentation: Release 0.4.25*. Aug. 2018.

[59] Melanie Swan. *Blockchain: Blueprint for a New Economy*. Ed. by Tim McGovern. First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., Feb. 2015. ISBN: 9781491920497.

[60] *The Growth of Blockchain Technology*. stackoverflow.

[61] *Top 100 Tokens By Market Capitalization*. CoinMarketCap. Aug. 2018. URL: https://coinmarketcap.com/tokens/.

[62] *Verordnung über die Vermeidung und Verwertung von Verpackungsabfällen (Verpackungsverordnung - VerpackV)*. Bundesgesetzblatt Teil I 56/1998. Bundesanzeiger Verlag, Aug. 1998.

[63] *Verordnung über die Vermeidung von Verpackungsabfällen (Verpackungsverordnung - VerpackV)*. Bundesgesetzblatt Teil I 36/1991. Bundesanzeiger Verlag, June 1991.

[64] Cora Wacker-Theodorakopoulos. "Pflichtpfand: Wirkungsloses Instrument". In: *Wirtschaftsdienst* 88.9 (2008), p. 558.

[65] Cora Wacker-Theodorakopoulos. *Zehn Jahre Duales System Deutschland*. 2000.

[66] Sha Wang and Jean-Philippe Vergne. "Buzz Factor or Innovation Potential: What Explains Cryptocurrencies' Returns?" In: *PLOS ONE* 12.1 (Jan. 2017), pp. 1–17.

[67] *White Paper: A Next-Generation Smart Contract and Decentralized Application Platform*. GitHub. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[68] Rachel Wolfson. *Vitalik Buterin On The State Of Ethereum, The Future Of Blockchain And Google Trying To Hire Him*. Forbes. Aug. 2018. URL: https://www.forbes.com/sites/rachelwolfson/2018/08/15/vitalik-buterin-on-the-state-of-ethereum-the-future-of-blockchain-and-google-trying-to-hire-him/amp/.

[69] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger Byzantium Version*. GitHub. June 2018.

[70] Dylan Yaga et al. *Blockchain Technology Overview*. National Institute of Standards and Technology. Jan. 2018.

Glossary

beverage packaging

Predominantly closed packaging for foods of liquid nature intended for consumption as a drink, excluding yogurt and kefir [63, § 3].

counterparty risk

Risk that the other party in a transaction will fail to meet their obligations [25, p. 175].

ecologically advantageous packaging

Packaging that does now show any significant ecological disadvantages when compared to reusable packaging [39, pp. 83].

nonce

Generally, a value that can only be used once. Here: number of confirmed transactions that have originated from an account [25, pp. 17, 148].

reusable packaging

Packaging intended to be reused for the same purpose after having been used. Characterised by having set up the logistics to take back, clean and refill the packaging. The sole intention or claim to be reused is not valid [63, § 3].

Appendices

A. Theoretical Framework

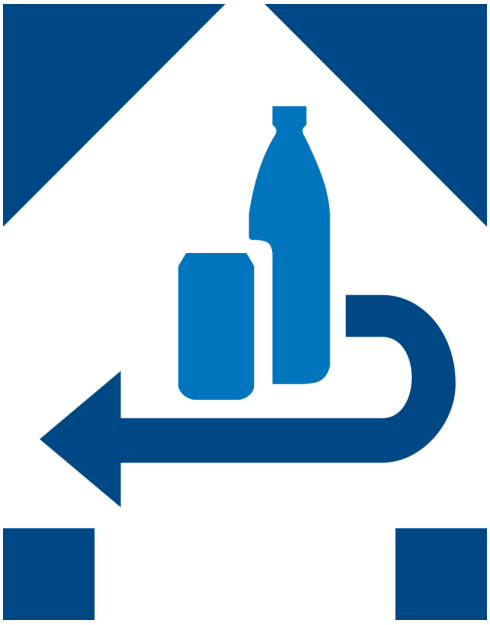

Figure A.1.: Universal security mark [36]

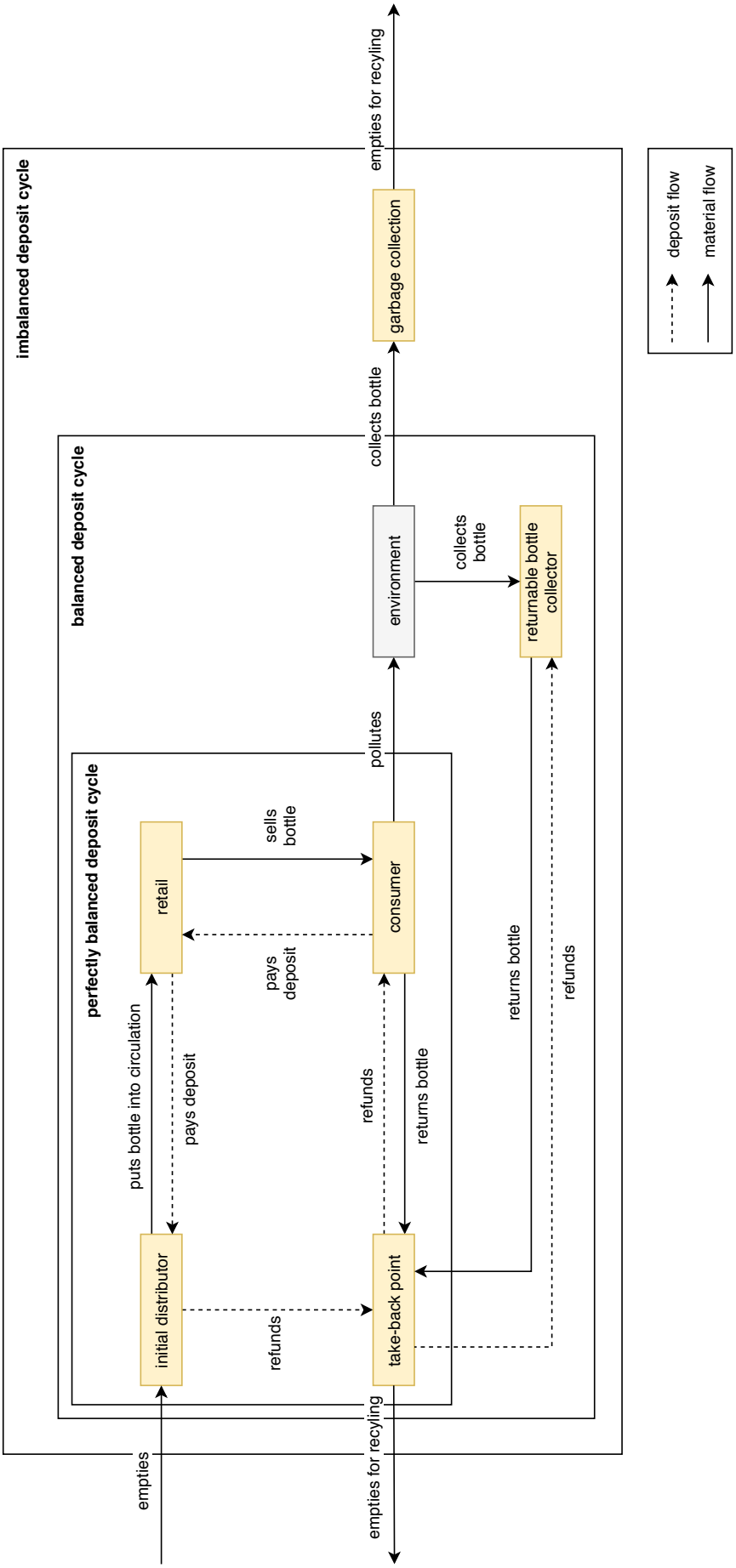

Figure A.2.: Deposit-refund cycle (extended)

B. Concept

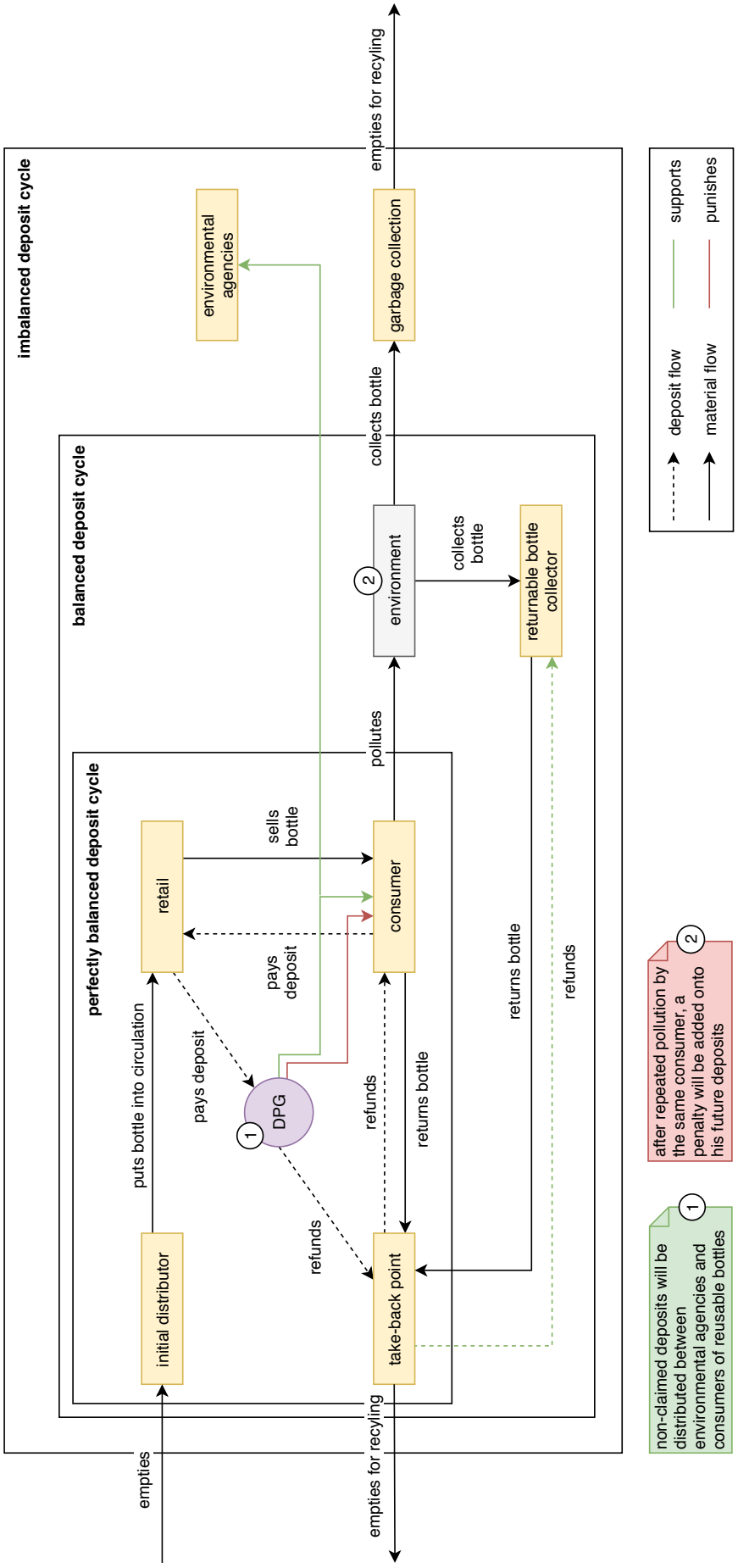

Figure B.1.: Future deposit-refund cycle (extended)

C. Implementation

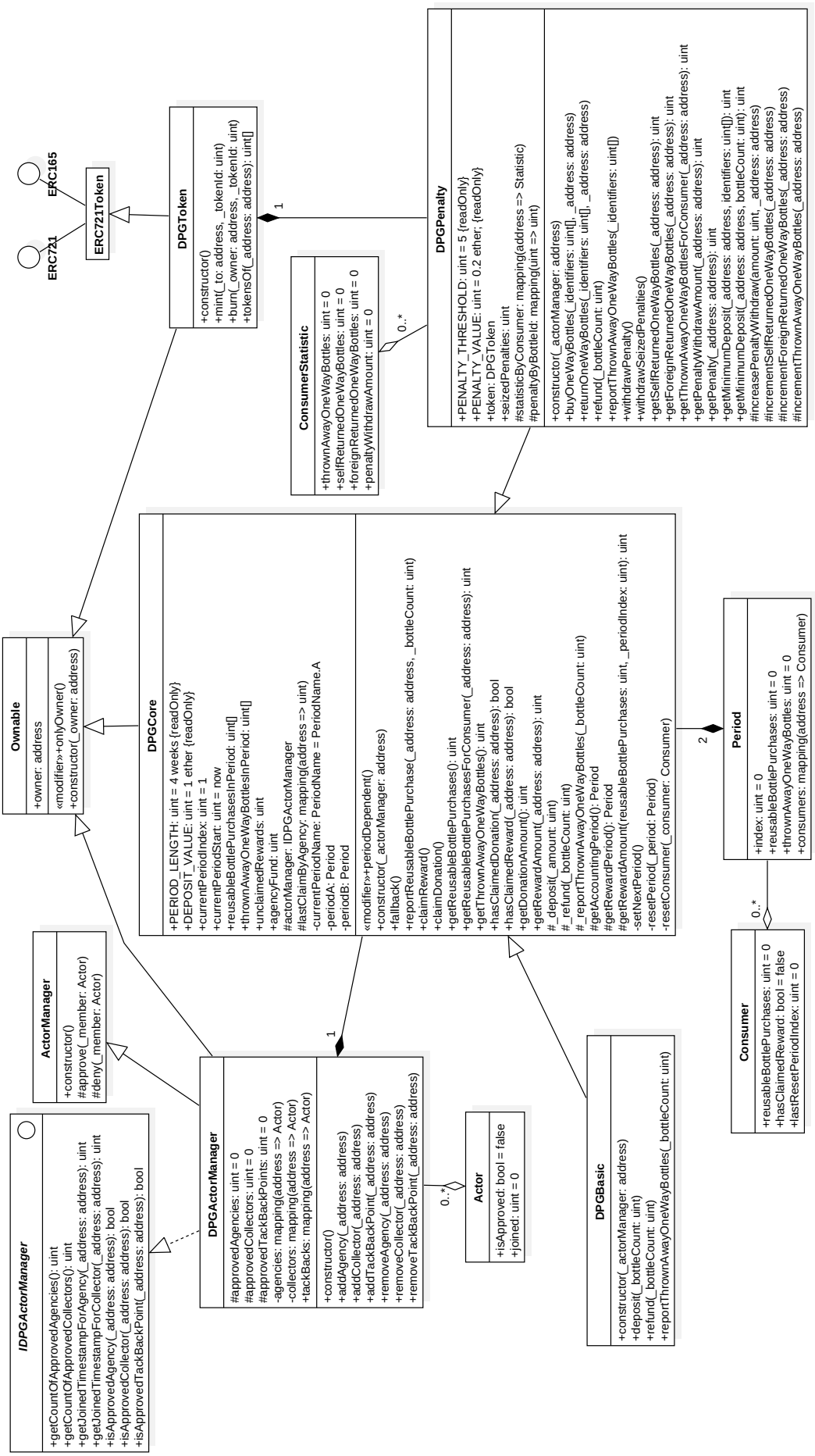

Figure C.1.: Class implementation model