Design of Blockchain-Based Apps Using Familiar Software Patterns with a Healthcare Focus

Peng Zhang, Vanderbilt University, Nashville, TN
Jules White, Vanderbilt University, Nashville, TN
Douglas C. Schmidt, Vanderbilt University, Nashville, TN
Gunther Lenz, Varian Medical Systems, Palo Alto, CA

Additional Key Words and Phrases: Software Patterns, DApps, Blockchain, Healthcare, Interoperability

ACM Reference Format:

Zhang, P., White, J., Schmidt, D. C., and Lenz, G. 2017. Design of Blockchain-Based Apps Using Familiar Software Patterns with a Healthcare Focus. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 14 pages.

1. INTRODUCTION

Over the past several years, blockchain technology has attracted interest from computer scientists and domain experts in various industries, including finance, real estate, healthcare, and transactive energy. This interest initially stemmed from the popularity of Bitcoin [Nakamoto 2012], which is a cryptographic currency framework that was the first application of blockchain. Blockchain possesses certain properties, such as decentralization, transparency, and immutability, that have allowed Bitcoin to become a viable platform for "trustless" transactions [Blundell-Wignall 2014], which can occur directly between any parties without the intervention of a trusted intermediary.

Another blockchain platform, Ethereum, extended the capabilities of the Bitcoin blockchain by adding support for "smart contracts" [Buterin 2014]. Smart contracts are code that directly controls the exchanges or redistributions of digital assets between two or more parties according to certain rules or agreements established between involved parties. Ethereum's programmable "smart contracts" enable the development of decentralized apps (DApps) [Johnston et al. 2014], which are autonomously operated services with data and records of operations cryptographically stored on the blockchain. DApps also enable direct interactions between end users and data on the blockchain.

Blockchain and its programmable smart contracts are being explored as a potential solution to address healthcare interoperability issues [DeSalvo and Galvez 2015; Das 2017]. Interoperability is defined as the ability for different information technology systems and software apps to communicate, exchange data, and effectively use the exchanged information [Geraci et al. 1991]. In the healthcare domain, it is necessary to achieve both syntactic

This work is sponsored in part by funding from Varian Medical Systems.
Author's address: P. Zhang, 1025 16th Ave S, Nashville, TN 37212; email: peng.zhang@vanderbilt.edu; J. White, 1025 16th Ave S, Nashville, TN 37212; email: jules.white@vanderbilt.edu; D. C. Schmidt, 1025 16th Ave S, Nashville, TN 37212; email: d.schmidt@vanderbilt.edu; G. Lenz, 3100 Hansen Way, Palo Alto, CA 94304; email: gunther.lenz@varian.com

and semantic interoperability, such as enabling effective interactions between users and medical applications, delivering patient data securely across healthcare facilities, and improving the overall efficiency of medical communications [Weininger et al. 2016]. Despite the interest in using blockchain technology for healthcare, however, little information is available on the concrete approaches for designing blockchain-based apps targeting healthcare-specific challenges.

This paper focuses on addressing this unexplored research topic: *providing recommendations for designing blockchain-based healthcare DApps using familiar software patterns to help mitigate healthcare challenges*. The target audience of this paper is thus healthcare IT system developers interested in applying blockchain technologies. Design patterns provide general solutions without tying specifics to a particular problem, allowing developers to communicate using well-known and well-understood names for software interactions [Shvets 2015]. By identifying these recurring patterns applicable to healthcare, this paper intends to help the target audience more quickly adapt to this technology and create robust healthcare solutions with it.

The remainder of this paper is organized as follows: Section 2 gives an overview of blockchains and the open source Ethereum blockchain implementation; Section 3 outlines key challenges regarding healthcare interoperability faced by direct applications of blockchains; Section 4 provides an end-to-end case study to illustrate the applications of four familiar software patterns to address interoperability requirements in the context of blockchain-based healthcare DApps; Section 5 summarizes existing research related to our work; and Section 6 concludes the paper and summarizes our future work on applying blockchain technologies in the healthcare domain.

2. OVERVIEW OF BLOCKCHAIN AND ITS ROLE IN HEALTHCARE APPS

This section gives a general overview of blockchain and the open-source Ethereum implementation that provides additional support for smart contracts, which are computer protocols that enable different types of decentralized apps beyond cryptocurrencies. The general blockchain overview is followed by a discussion of Solidity, which is a programming language used to write Ethereum smart contracts.

2.1 Overview of Blockchain

A blockchain is a decentralized computing architecture that maintains a growing list of ordered transactions grouped into blocks that are continually reconciled to keep information up-to-date, as shown in Figure 1. All

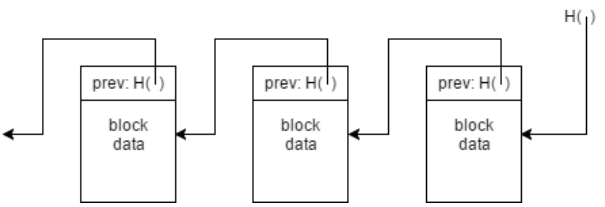

Fig. 1. Blockchain Structure: a Continuously Growing List of Ordered and Validated Transactions

transaction records are kept in the blockchain and are shared with all network nodes. Only one block can be added to the blockchain at a time. Each block is mathematically verified (using cryptography) to ensure that it follows in sequence from the previous block. The verification process is called "mining" or *Proof of Work* [Nakamoto 2012], which allows network nodes (also called "miners") to compete to have their block be the next one added to the blockchain by solving a computationally expensive puzzle. The winner then announces the solution to the entire network to gain a mining reward paid via cryptocurrency. The mining process combines cryptography, game theory, and incentive engineering to ensure that the network reaches consensus regarding each block in the blockchain and that no tampering occurs in the transaction history. This process ensures properties of

transparency, immutability, and decentralization (resilient to a single point of failure due to replicated storage) [Hub 2017].

In the Bitcoin application, blockchain serves as a public ledger for all cryptocurrency transactions in bitcoins to promote trustless financial exchanges between individual users, securing all their interactions with cryptography. The Bitcoin blockchain has limitations, however, when supporting different types of applications involving contracts, equity, or other information, such as crowdfunding, identity management, and democratic voting registry [Buterin 2014]. To address the needs for a more flexible framework, Ethereum was created as an alternative blockchain, giving users a generalized trustless platform that can run smart contracts.

The Ethereum blockchain is a distributed state transition system, where state consists of accounts and state transitions are direct transfers of value and information between accounts. Two types of accounts exist in Ethereum: (1) *externally owned accounts* (EOAs), which are controlled via private keys and only store Ethereum's native value-token "ether" and (2) *smart contract accounts* (SCAs) that are associated with contract code and can be triggered by transactions or function calls from other contracts [Buterin 2014].

To protect the blockchain from malicious attacks and abuse (such as distributed denial of service attacks in the network or hostile infinite loops in smart contract code), Ethereum also enforces a payment protocol, whereby a fee (in terms of "gas" in the Ethereum lexicon) is charged for data storage and each data operation executed in a contract. These fees are collected by miners who verify, execute, propagate transactions, and then group transactions into blocks.

As in the Bitcoin network, the mining rewards provide an economic incentive for users to devote powerful hardware and electricity to the public Ethereum network. In addition, transactions have a gas limit field to specify the maximum amount of gas that the sender is willing to pay for. If gas used during transaction execution exceeds this limit, computation is stopped, but the sender still has to pay for the performed computation. This protocol also protects senders from completely running out of funds.

2.2 Overview of Solidity

Ethereum smart contracts can be built in a Turing-complete programming language, called Solidity [Foundation 2015b]. This contract language is compiled by the Ethereum Virtual Machine (EVM), which enables the Ethereum blockchain to become a platform for creating DApps that provide potential solutions to certain healthcare interoperability challenges. Solidity has an object-oriented flavor and is intended primarily for writing contracts in Ethereum.

A "class" in Solidity is realized through a "contract," which is a prototype of an object that lives on the blockchain. Just like an object-oriented class can be instantiated into a concrete object at runtime, a contract may be instantiated into a concrete SCA by a transaction or a function call from another contract. At instantiation, a contract is given a uniquely identifying address, similar to a reference or pointer in C/C++-like languages, with which it can then be called.

Contracts may contain persistent state variables that can be used as data storage and functions that interact with the states. Although one contract can be instantiated into many SCAs, it should be treated as a singleton to avoid storage overhead. After a contract is created, its associated SCA address is typically stored at some place (*e.g.*, a configuration file or a database) and used as a parameter by an app to access its internal states and functions [Dourlens 2017].

Solidity supports multiple inheritance and polymorphism [Ethereum.io 2017]. When a contract inherits from one or more other contracts, a single contract is created by copying all the base contracts code into the created contract instance. Abstract contracts in Solidity allow function declaration headers without concrete implementations. They cannot be compiled into an SCA but can be used as base contracts. Due to Solidity contracts' similarity to C++/Java classes, certain software patterns can be directly applied to smart contracts as well, as we describe in Section 4.

3. HEALTHCARE INTEROPERABILITY CHALLENGES FACED BY BLOCKCHAIN-BASED APPS

Many research and engineering ideas have been proposed to apply blockchains to healthcare and implementation attempts are underway [Ekblaw et al. 2016; Peterson et al. 2016; Porru et al. 2017; Bartoletti and Pompianu 2017]. Few published studies, however, have addressed software design considerations needed to implement blockchain-based healthcare apps effectively. While it is crucial to understand the fundamental properties of blockchains, it is also important to apply them properly so that healthcare-specific challenges are addressed. The US Office of the *National Coordination for Health Information Technology* (ONC) has outlined basic technical requirements for achieving interoperability [ONC 2014]. This section summarizes key interoperability challenges faced by blockchain-based apps, focusing on four aspects: system evolvability, costly storage on blockchain, information privacy, and scalability.

3.1 Evolvability Challenge: Maintaining Evolvability While Minimizing Integration Complexity

Many apps are written with the assumption that data is easy to change. However, once stored in the blockchain, data is difficult to modify *en masse* and its change history is immutable. A critical design consideration when building blockchain apps for healthcare is thus ensuring that the data written into blockchain contracts are designed to facilitate evolution where needed. Although evolution must be supported, healthcare data must often be accessible from a variety of deployed systems that cannot easily be changed over time. Apps should therefore be designed in a way that is loosely coupled and minimizes the usability impact of evolution on the clients, *i.e.*, user services that interact with data in the blockchain. Section 4.2.1 shows how using the *Abstract Factory* pattern can facilitate evolution, while minimizing the impact on dependent clients, focusing on a specific evolution challenge involving entity creation and management of healthcare participants.

3.2 Storage Challenge: Minimizing Data Storage Requirements

Healthcare apps can serve thousands or millions of participants, which may incur enormous overhead when *large* volumes of data are stored in a blockchain–particularly if data normalization and denormalization techniques are not carefully considered. Not only is it costly to store these data, but data modification and access operations may also fail if/when the cost exceeds the Ethereum gas limit discussed in Section 2.1. An important design consideration for blockchain-based healthcare apps is thus to minimize data storage requirements in addition to provide sufficient flexibility to manage individual health concerns. Section 4.2.2 shows how to design smart contracts with the *Flyweight* pattern to ensure account uniqueness and maximize common intrinsic data sharing across accounts while still allowing extrinsic data to vary in specific individual accounts.

3.3 Privacy Challenge: Balancing Data Storage with Privacy Concerns

Blockchains and smart contracts can offer trustless digital health asset sharing, audit trails of data access, and decentralized and replicated storage, which are essential for improving healthcare interoperability by providing ubiquitous data store. Although there are substantial potential benefits to data availability if stored in the blockchain, there are also significant risks due to the transparency of blockchain. In particular, even when encryption is applied it is still possible that the current encryption techniques may be broken in the future or that vulnerabilities in the encryption implementations used may lead to private information potentially being decryptable in the future. Section 4.2.3 shows how designing a blockchain-based app using the *Proxy* pattern can facilitate interoperability while keeping sensitive patient data from being directly encoded in the blockchain.

3.4 Scalability Challenge: Tracking Relevant Health Changes Scalably Across Large Patient Populations

Communication gaps and information sharing challenges are serious impediments to healthcare innovation and the quality of patient care. Providers, hospitals, insurance companies, and departments within health organizations experience disconnectedness caused by delayed or lack of information flow. Patients are commonly cared for by various sources, such as private clinics, regional urgent care centers, and enterprise hospitals. A provider

may have hundreds or more patients whose associated health data must be tracked. Section 8 shows how a blockchain-based app design using the *Publisher-Subscriber* pattern can be aid in scalably detecting and communicating relevant health changes.

4. DESIGNING BLOCKCHAIN-BASED HEALTH APPS WITH FAMILIAR DESIGN PATTERNS

This section presents the structure and functionality of the *DApp for Smart Health* (DASH)[1] we are developing to explore the efficacy of applying blockchain technology to the healthcare domain. It then describes applications of familiar software patterns in blockchain-based health apps (such as DASH or other DApps) to address the interoperability challenges described in Section 3.

4.1 DASH Overview

Implemented on an Ethereum test blockchain, DASH provides a web-based portal for patients to access and update their medical records, as well as submit prescription requests. Likewise, DASH allows providers to review patient data and fulfill prescription requests based on permissions given by patients. Figure 2 shows the structure and workflow of DASH.

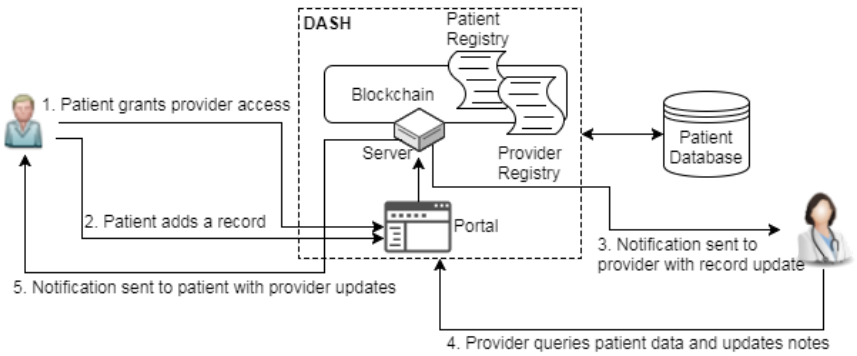

Fig. 2. Structure and Workflow of DASH

DASH uses a *Patient Registry* contract to store a mapping between unique patient identifiers and their associated *Patient Account* contract addresses. Each *Patient Account* contract contains a list of health providers that are granted read/write access to the patient's records (*i.e.*, medical records and prescription history). Currently, DASH provides basic access services to two types of users: patients and providers. The high-level workflow of DASH is as follows:

(1) Patient first specifies a provider's read/write access of their records using the DASH portal

(2) Patient then adds a health record or sends a prescription request

(3) DASH server captures the patient health activities and notifies their specified care provider

(4) Provider can then use the portal to access the patient record or handles their prescription requests

(5) DASH server notifies the patient with associated provider activity

[1] There is no relationship between our DASH app and the Dash cryptocurrency, even though they are both based on blockchain technologies.

4.2 Integrating Familiar Software Patterns to DApp Designs

The remainder of this section applies a pattern form variant to motivate and show how familiar software patterns are manifest in blockchain-based healthcare apps. In particular, we focus on four software patterns—*Abstract Factory*, *Flyweight*, *Proxy*, and *Publisher-Subscriber* [Gamma et al. 1995; Buschmann et al. 2007]. We explain how these patterns are incorporated in DASH and describe the forces that they resolve in the blockchain platform[2].

Figure 3 presents how DASH applies software patterns to address the following design challenges:

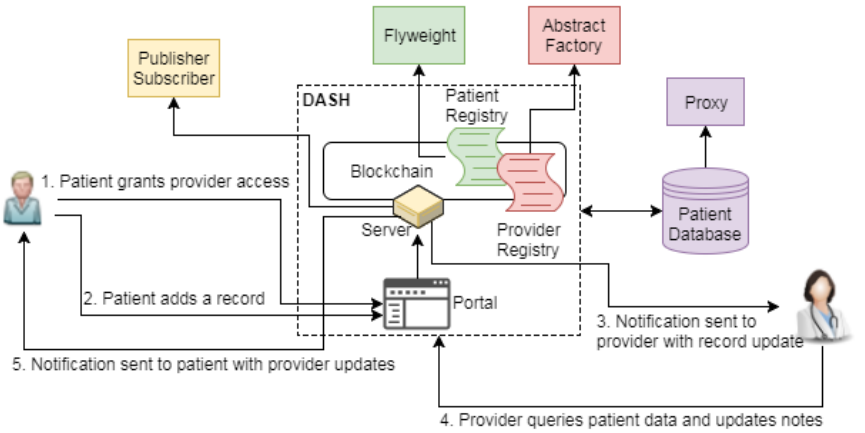

Fig. 3. Incorporating Software Patterns to the Design of DASH

—*Abstract Factory* assists with user/entity account creation and management based on user types (*e.g.* in the current implementation of DASH, provider and patient).

—*Flyweight* ensures unique account creation on the blockchain and maximizes sharing of common, intrinsic data (*e.g.* in the example of DASH, information that is common across patients).

—*Proxy* protects health information privacy with a simpler surrogate object that has the ability to interact with other components in the system (*e.g.* proxifying real patient database in DASH).

—*Publisher-Subscriber* aids in scalably managing health change events and actively notifying healthcare participants only when relevant changes occur (*e.g.* DASH server's notification service).

Applications of these patterns to address the technical challenges from Section 3 are discussed in depth below.

4.2.1 *Entity Creation with Abstract Factory.*

Design problem faced by blockchain-based apps. If a blockchain-based healthcare app needs to create entity/user accounts for an evolving, hierarchical organization (such as a hospital group), one design problem is how to maintain evolvability and ease of integration when new entities of different functions (*e.g.*, a new division or department) are introduced. In particular, the immutability property of blockchain ensures that interfaces (*e.g.*, methods and state structure) of already instantiated contracts cannot be modified or upgraded. As a result, each subsequent version of a smart contract must be created as a new contract object on the blockchain, and data stored in the previous versions of the contract must be copied to the new version. Both steps incur transaction fees that must be paid to the blockchain miners. To avoid extra costs and repeated data copying operations, it is therefore important to design a smart contact that modularizes code and minimizes interface changes over time.

[2]Naturally, there are other patterns relevant in this domain, which we will focus on in future work.

For example, if a contract has a function that facilitates interactions between different departments in a highly structured hospital, without a coherent design many decisions must be made to find the appropriate departmental accounts involved (a large number of branching statements). As new departments are introduced, the decision-making code will likely have to change more than once with each obsolete version of contract discarded. A desired design should minimize interface changes to a contract.

Solution → Apply the *Abstract Factory* pattern to improve design evolvability. Creating entities/accounts in smart contracts for an evolving, hierarchical organization can benefit from applying the *Abstract Factory* pattern [Gamma et al. 1995]. This pattern allows DApps like DASH to shift the responsibility for providing entity/account creation services to an abstract "factory" object [Gamma et al. 1993] (the factory itself is a contract instance). A concrete factory object can inherit methods from the abstract factory and customize them to create accounts for a specific organization/department/group of related or interacting sub-entities. When a new department is introduced, it is easy to create another concrete department factory object without affecting existing contracts.

Figure 4 shows the basic structure of this pattern in the context of DASH. In this design, *AbstractAccountFactory*,

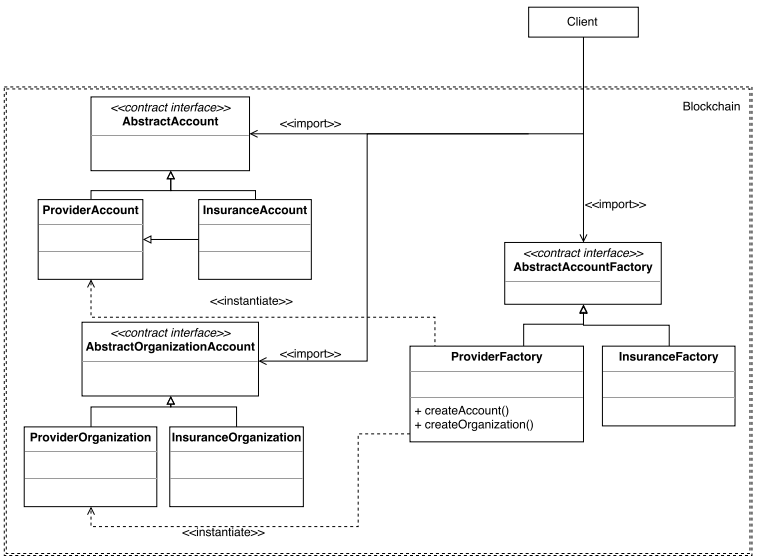

Fig. 4. Entity Creation with *Abstract Factory* in DASH

the abstract factory object, defines some common logic (*i.e.*, *createAccount* and *createOrganization* methods) shared by all concrete factories (*e.g.*, *ProviderFactory*, *InsuranceFactory*, etc). It allows these concrete factories to create families of related accounts, such as a *Provider* account and a *Provider Organization* account for the *Provider* entity. The process of creating a new entity (*e.g.*, pharmacy) in this design is completely decoupled from all other components, leaving the existing contracts intact.

Consequences of applying *Abstract Factory*. Without using a high-level abstraction, such as an abstract factory, clients interested in creating constituent accounts for an entity must access each concrete account creation factory and make many if-else decisions at runtime. This tight coupling is cumbersome since clients need to know the implementation details well to use each factory's methods correctly. For example, to create accounts for the *provider* and *insurance* entities, the client has to import both *ProviderFactory* and *InsuranceFactory* contracts and then invoke the appropriate methods (*e.g.*, *createAccount()* and *createOrganization() for the provider entity* for each account creation, as shown in Figure 5. As an entity/organization scales out and up with more components/departments, clients will be overwhelmed with the implementation details.

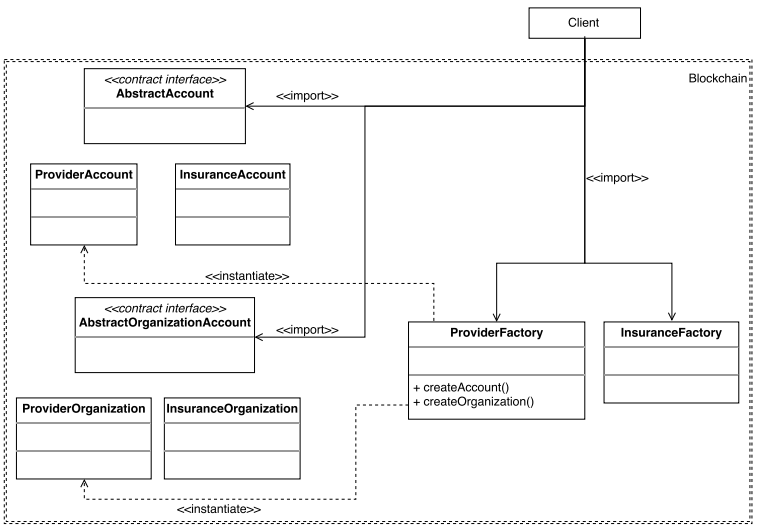

Fig. 5. Without the Application of *Abstract Factory* Pattern to DASH

Entity creation with *Abstract Factory* introduces a weak coupling between the client (*i.e.*, DApp) and specific contract implementations. In particular, newly defined entity interactions can inherit from the abstract contract to preserve common properties and have entity-specific customizations while leaving existing contracts unmodified to avoid contract deprecation. The downside of using this design in a blockchain, however, is the extra cost incurred by an added layer of indirection in terms of storage for the abstract factory and computation for its instantiation.

4.2.2 *Data Sharing with Flyweight Registry.*

Design problem faced by blockchain-based apps. All data and transaction records maintained in the blockchain are replicated and distributed to every node in the network. To compensate blockchain miners for contributing expensive hardware to store and maintain the data, fees are charged based on the storage requirement of an application. To minimize storage cost overhead, a blockchain-based healthcare app that requires storage of some data on-chain must maximize data sharing among entities thus limit the amount of information stored.

In a large-scale healthcare scenario, if a blockchain is used to store patient billing data, there will be millions of records replicated on all blockchain miner nodes. Moreover, billing data could include detailed patient insurance information, such as their ID#, insurance contact information, coverage details, and other aspects that the provider needs to bill for services. Capturing all this information for every patient can generate excessive amounts of data in the blockchain.

Suppose it is necessary to store some insurance and billing information (encrypted) in the blockchain. Most patients are covered by one of a relatively small subset of insurers (in comparison to the total number of patients, *e.g.*, each insurance policy may cover 10,000s or 100,000s of patients). Therefore, a substantial amount of intrinsic, non-varying information is common across patients that can be reused and shared, such as details on what procedures are covered by an insurance policy. To bill for a service, however, this common intrinsic information must be combined with extrinsic information (such as the patient's ID#) that is specific to each patient.

A good design should maximize sharing of such common data to reduce on-chain storage cost and meanwhile have the capability to provide complete data objects on demand.

Solution → Apply the *Flyweight* pattern to minimize data storage in the blockchain. Combining the *Flyweight* pattern [Gamma et al. 1995] with a factory [Gamma et al. 1993] can help minimize data storage in the blockchain. In particular, the factory can create a **registry model** that stores shared data in a common contract,

i.e., the registry, while externalizing varying data storage in entity-specific contracts. The registry can keep track of references (*i.e.*, addresses) to entity-specific contracts and retrieve combined extrinsic and intrinsic data upon request.

Figure 6 shows the flyweight registry model applied to DASH. By creating a registry model in DASH, shared

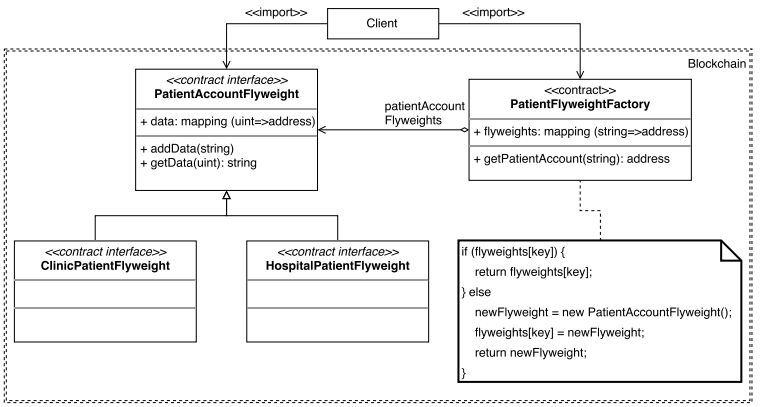

Fig. 6. Applying the Flyweight Registry Model to DASH

patient insurance information is stored only once in the registry, avoiding an exorbitant amount of memory usage from saving repeated data in all patient accounts. Varying, patient-specific billing information is stored in corresponding patient-specific account contracts.

The registry in DASH also maintains a mapping between unique user identifiers and the referencing addresses of patient account contracts to prevent account duplication. At account creation, only if no account with the specified patient identifier exists in the registry does it create a new account; otherwise the registry retrieves the address associated with the existing account contract. To retrieve complete insurance and billing information of a particular patient, clients need only invoke a function call from the registry with the patient identifier to obtain the combined intrinsic and extrinsic data object.

Consequences of applying the *Flyweight* pattern. The flyweight registry model provides better management for the large pool of objects (such as user accounts in the example above). It minimizes redundancy in similar objects by maximizing data and operation sharing. Particularly in the insurance example, if common insurance policy details are not extracted from each patient's contract, the cost to change a policy detail will be immense–it will require rewriting a huge number of impacted contracts. Data sharing with flyweight registry helps minimize the cost to change the common state in objects stored on-chain.

Although applying the *Flyweight* pattern creates an additional transaction to verify and include in the blockchain (*i.e.*, the flyweight object instantiation) before it can be used, this extra step can be outweighed by the resulted efficiency in data management.

4.2.3 *Privacy Protection with Proxy and Oracle.*

Design problem faced by blockchain-based apps. If a blockchain-based healthcare app must expose sensitive data or metadata (such as patient identifying information) on the blockchain, it must be designed to maximize health data privacy while facilitating health information exchange. In particular, a fundamental aspect of a blockchain is that data and all change history stored on-chain are public, immutable, and verifiable. For financial transactions focused on proving that transfer of an asset occurred, these properties are critical. When the goal is to store data in the blockchain, however, it is important to understand how these properties will impact the use case.

For example, storing patient data in the blockchain can be problematic since it requires that data be public and immutable. Although data can be encrypted before being stored, should all patient data be publicly distributed to all blockchain nodes? Even if encryption is used, the encryption technique may be broken in the future or defects in the implementation of the encryption algorithms or protocols used may make the data decryptable in the future. Immutability, on the other hand, prevents owners of the data from removing the data change history from the blockchain if a security flaw is found. Many other scenarios, ranging from discovery of medical mistakes in the data to changing data standards may necessitate the need to change the data over time.

In scenarios where the data may need to be changed, the public and immutable nature of the blockchain creates a fundamental tension that must be resolved. On the one hand, healthcare providers would like incorruptible data so its integrity is preserved. At the same time, providers want the data changeable and secure to protect patient privacy and account for possible errors. An interoperable app should protect patient privacy and also ensure data integrity.

Solution → Apply the *Proxy* pattern to enable secure and private data services. Combining the *Proxy* pattern [Gamma et al. 1995] with a data retrieving service (such as Oracle [Foundation 2015a]) can enable secure and private data services. The Oracle network is a third-party service that allows a contract to query/retrieve data sources outside the blockchain address space and ensures data retrieved is genuine and uncompromised [Hub 2017]. To reduce computational costs on-chain, a proxy can be created to provide some lightweight representation or placeholder for the real data object until its retrieval is required.

Figure 7 shows the application of a proxy in DASH. DASH uses a proxy contract to expose some simple

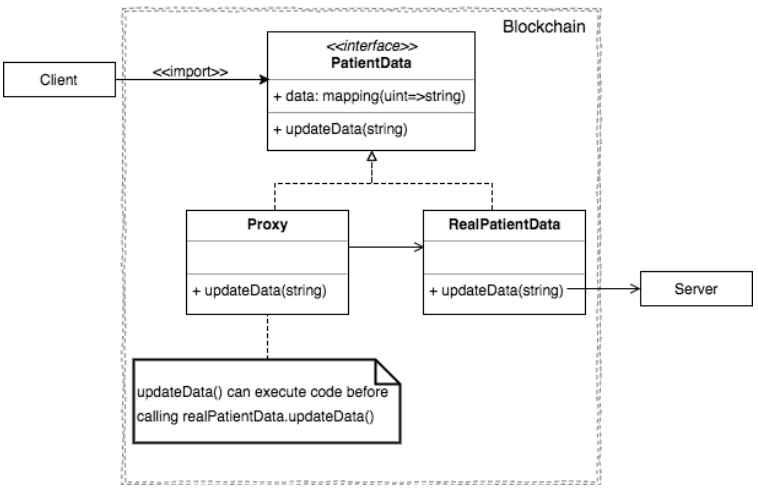

Fig. 7. Structure of a Proxy Application in DASH

metadata about a patient and later refer to the heavyweight implementation on demand to obtain the complete data via an Oracle. Each read request and modification operation can be logged in an audit trail that is transparent to the entire blockchain network for verification against data corruption. In the case of a proxified contract (heavyweight implementation) being updated with a new storage option (*e.g.*, replacing an Oracle with some other data service), the interface to the proxy contract can remain unchanged, encapsulating low-level implementation variations.

Consequences of applying the *Proxy* pattern. A proxy object can perform lightweight housekeeping or auditing tasks by storing some commonly used metadata in its internal states without having to perform expensive operations (such as retrieving health data from an Oracle service). It follows the same interface as the real object

and can execute the original heavyweight function implementations as needed. It can also hide information about the real object from the client to protect patient data privacy.

However, *Proxy* may cause disparate behavior when the real object is accessed directly by some client while the proxy surrogate is accessed by others. Nonetheless, proper usage of a proxy with an Oracle service can provide a channel for protected information exchange on the blockchain.

4.2.4 *Tracking Changes with a Notification Service.*

Design problem faced by blockchain-based apps. A blockchain-based healthcare system that needs to track relevant health changes across large patient populations must be designed to filter out useful health-related information from communication traffic (*i.e.* transaction records) in the blockchain. For example, the Ethereum blockchain maintains a public record of contract creations and operation executions along with regular cryptocurrency transactions. The availability of this information makes blockchain a more autonomous approach to improve the coordination of patient care across different participants (*e.g.*, physicians, pharmacists, insurance agents, etc) who would normally communicate through various channels with a lot of manual effort, such as through telephoning or faxing. Due to the continually growing list of records on the blockchain, however, directly capturing any specific health-related topic from occurred events implies exhaustive transaction receipt lookups and topic filtering, which requires non-trivial computation and may result in delayed responses.

A good model should facilitate coordinated care and support relevant health information relays. For instance, care should be seamless from the point when a patient self-reports illness (through a health DApp interface) to the point when they receive the prescriptions created by their primary care provider; clinical reports and follow-up procedure should be relayed to and from the associated care provider offices in a timely manner.

Solution → Apply the *Publisher-Subscriber* pattern to facilitate scalable information filtering. Incorporating a notification service using the *Publisher-Subscriber* pattern [Buschmann et al. 1996] can facilitate scalable information filtering. In this design, changes in health activities are only broadcast to providers that subscribe to events relating to their patients. It alleviates tedious filtering of which care provider should be notified about patient activities as large volumes of transactions take place. It also helps maintain an interoperable environment that allows providers across various organizations to participate. To avoid computation overhead on the blockchain, the actual processing of patient activities data can be done off-chain by a DApp server.

Specifically, when the publisher sends an update, its subscribers only need to do a simple update to an internal state variable that records the publisher's address, which the DApp server actively monitors for change. When a change occurs, the responsibility for the heavy computational content filtering task (*e.g.,* retrieving the change activity from the publisher using the address) is delegated to the DApp server from the blockchain. The DApp server is context-aware at this point because each subscriber has an associated contract address accessible by the server. The server can then filter the content based on subscribed topics and update the contract states of appropriate subscribers as needed.

Figure 8 shows the design of DASH using the *Publisher-Subscriber* pattern for the notification service. In DASH, events associated with patient-reported sickness symptoms (a concrete *Topic*) are subscribed to by the patient's primary care provider and pharmacist. When this event occurs, the *PatientPublisher* contract notifies the *Subscriber*s by updating the *state* value in the concrete subscriber contracts. DASH server, which actively monitors for changes in the subscriber states, finds the updated status and in turn queries the latest patient health changes to pass onto the proper provider subscribers.

Consequences of applying the *Publisher-Subscriber* pattern. Applying a notification service in a healthcare DApp design is useful when a state change in one contract must be reflected in others without keeping the contracts tightly coupled. Adding/removing subscribers and topics is trivial as it only requires minimal changes in the state variables. It also makes topic/content filtering more manageable when subscription relations are clearly defined in each participant's contract state. In addition, this design enables communication across participants from various organizations, as required of an interoperable system.

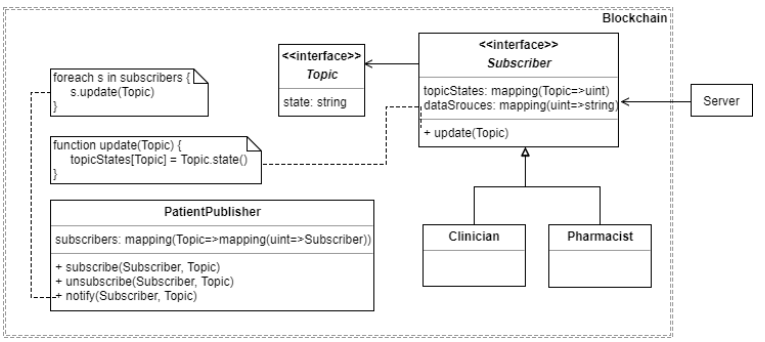

Fig. 8. Using the *Publisher-Subscriber* Pattern to Provide Notification Service in DASH

The limitations of using a notification service on the blockchain include (1) delays in updates received by subscribers due to the extra step of validation required by the blockchain infrastructure and (2) high storage costs associated with defining very fine-grained topic subscriptions on-chain.

Concretely, in certain blockchain implementations, such as Bitcoin, the order of which transactions are to be executed and verified is determined by the miners based on the transaction fees paid by the senders and how long transactions have been in the transaction pool [Kasahara and Kawahara 2016]. Transactions with the highest priority will be executed first and added to the blockchain sooner, which could cause unexpected delays in the notification service.

If subscribers want to receive fine-grained message topics, the amount of computation for filtering out the messages sent by publishers may be prohibitively expensive. This overhead may result in either failure to publish messages due to gas shortage or unacceptable implementation costs. One solution is to have broader topics with fewer filter requirements on-chain and handle more detailed message filtering off the blockchain.

5. RELATED WORK

Although relatively few papers focus on realizing software patterns in blockchains, some relate to healthcare blockchain solutions and design principles in this space. This section gives an overview of related research on (1) the challenges of applying blockchain-based technology in the healthcare space and innovative implementations of blockchain-based healthcare systems and (2) design principles and recommended practice for blockchain application implementations.

5.1 Challenges of healthcare blockchain and proposed solutions.

Ekblaw et al. [Ekblaw et al. 2016] proposed MedRec as an innovative, working healthcare blockchain implementation for handling EHRs, based on principles of existing blockchains and Ethereum smart contracts. The MedRec system uses database "Gatekeepers" for accessing a node's local database governed by permissions stored on the MedRec blockchain. Peterson et al. [Peterson et al. 2016] presented a healthcare blockchain with a single centralized source of trust for sharing patient data, introducing *Proof of Interoperability* based on conformance to the FHIR protocol as a means to ensure network consensus.

5.2 Applying software design practice to blockchain.

Porru et al. [Porru et al. 2017] highlighted evident challenges in state-of-the-art blockchain-oriented software development by analyzing open-source software repositories and addressed future directions for developing blockchain-based software. Their work focused on macro-level design principles such as improving collaboration, integrating effective testing, and evaluations of adopting the most appropriate software architecture. Bartoletti et al. [Bartoletti and Pompianu 2017] surveyed the usage of smart contracts and identified nine common software

patterns shared by the studied contracts, *e.g.*, using "oracles" to interface between contracts and external services and creating "polls" to vote on some question. These patterns summarize the most frequent solutions to handle some repeated scenarios.

6. CONCLUDING REMARKS

Blockchain and its programmable smart contracts provide a platform for creating decentralized apps that have the potential to improve healthcare interoperability. Leveraging this platform in a decentralized and transparent manner, however, requires that key design concerns be addressed. These concerns include—but are not limited to—system evolvability, storage requirements minimization, patient data privacy protection, and application scalability across large number of users. This paper described these concerns and showed how we are mitigating these challenges in our blockchain-based *DApp for Smart Health* (DASH) by applying familiar software patterns–*Abstract Factory*, *Flyweight*, *Proxy*, and *Publisher-Subscriber*.

Based on our experience developing the DASH case study, we learned the following lessons:

—The public, immutable, and verifiable properties of the blockchain enable a more interoperable environment that is not easily achieved using traditional approaches, which mostly rely on a centralized server or data storage.

—Each time a smart contract is modified, a new contract object is created on the blockchain. Important design decisions must therefore be made in advance to avoid the cost and storage overhead from contract interface change.

—To best leverage these properties of blockchain in the healthcare context, concerns regarding system evolvability, storage costs, sensitive information privacy, and application scalability must be taken into account.

—Combining time-proven design practices with the unique properties of the blockchain enables the creation of DApps that are more modular, easier to integrate and maintain, and less susceptible to change.

Our future work will extend the app described in Section 4 to delve into the challenges and pinpoint the most practical design process in creating a blockchain-based healthcare architecture. Furthermore, we will explore the potential application of other software patterns to handle various challenges, such as security, dependability, and performance, in addition to assessing the applications in other blockchain platforms such as the EOS asynchronous smart contract platform [block.one 2017]. One approach will evaluate the efficacy of applying these software patterns (*e.g.,* via performance metrics related to time and cost of computations or assessment metrics related to its feasibility) compared to other alternative designs (such as designs without using software patterns). We will also investigate extensions of the blockchain from a healthcare perspective, such as creating an alternative health chain that exclusively serves the healthcare sector.

REFERENCES

Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. *arXiv preprint arXiv:1703.06322* (2017).

block.one. 2017. EOS.IO Technical White Paper. *https://github.com/EOSIO/Documentation* (2017).

Adrian Blundell-Wignall. 2014. The Bitcoin question: Currency versus trust-less transfer technology. *OECD Working Papers on Finance, Insurance and Private Pensions* 37 (2014), 1.

Frank Buschmann, Kelvin Henney, and Douglas Schmidt. 2007. *Pattern-oriented Software Architecture: On Patterns and Pattern Languages*. Vol. 5. John Wiley & Sons.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-oriented Software Architecture: A System of Patterns*. Vol. 1. John Wiley & Sons.

Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. *https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper* (2014).

Reenita Das. 2017. Does Blockchain Have A Place In Healthcare. (2017). https://www.forbes.com/sites/reenitadas/2017/05/08/does-blockchain-have-a-place-in-healthcare/

K DeSalvo and E Galvez. 2015. Connecting health and care for the nation: a shared nationwide interoperability roadmap. *Health IT Buzz* (2015).

Jules Dourlens. 2017. Ethereum smart contracts lifecycle. (2017). https://ethereumdev.io/ethereum-smart-contracts-lifecycle/

Ariel Ekblaw, Asaph Azaria, John D Halamka, and Andrew Lippman. 2016. A Case Study for Blockchain in Healthcare: "MedRec" prototype for electronic health records and medical research data. (2016).

Ethereum.io. 2017. Contracts. (2017). http://solidity.readthedocs.io/en/develop/contracts.html

Ethereum Foundation. 2015a. ORACLIZE LIMITED. (2015). http://www.oraclize.it/

Ethereum Foundation. 2015b. Solidity. (2015). https://solidity.readthedocs.io/en/develop/

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 406–431.

Erich Gamma, John Vlissides, Ralph Johnson, and Helm. Richard. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Anne Geraci, Freny Katki, Louise McMonegal, Bennett Meyer, John Lane, Paul Wilson, Jane Radatz, Mary Yee, Hugh Porteous, and Fredrick Springsteel. 1991. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press.

Blockchain Hub. 2017. Blockchain Oracles. (2017). https://insights.sei.cmu.edu/sei_blog/2017/07/what-is-bitcoin-what-is-blockchain.html

David Johnston, Sam Onat Yilmaz, Jeremy Kandah, Nikos Bentenitis, Farzad Hashemi, Ron Gross, Shawn Wilkinson, and Steven Mason. 2014. The General Theory of Decentralized Applications, DApps. *GitHub, June* 9 (2014).

Shoji Kasahara and Jun Kawahara. 2016. Priority Mechanism of Bitcoin and Its Effect on Transaction-Confirmation Process. *arXiv preprint arXiv:1604.00103* (2016).

Satoshi Nakamoto. 2012. Bitcoin: A peer-to-peer electronic cash system, 2008. *http://www.bitcoin.org/bitcoin.pdf* (2012).

ONC. 2014. Connecting Health and Care for the Nation: A 10-Year Vision to Achieve an Interoperable Health IT Infrastructure. (2014).

Kevin Peterson, Rammohan Deeduvanu, Pradip Kanjamala, and Kelly Boles. 2016. A Blockchain-Based Approach to Health Information Exchange Networks. (2016).

Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. 2017. Blockchain-oriented software engineering: challenges and new directions. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 169–171.

Alexander Shvets. 2015. Design Patterns Explained Simply. *sourcemaking. com* (2015), 80–84.

Sandy Weininger, Michael B Jaffe, Michael Robkin, Tracy Rausch, David Arney, and Julian M Goldman. 2016. The importance of state and context in safe interoperable medical systems. *IEEE journal of translational engineering in health and medicine* 4 (2016), 1–10.