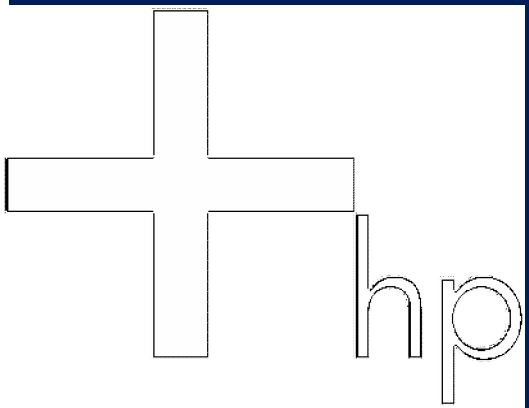
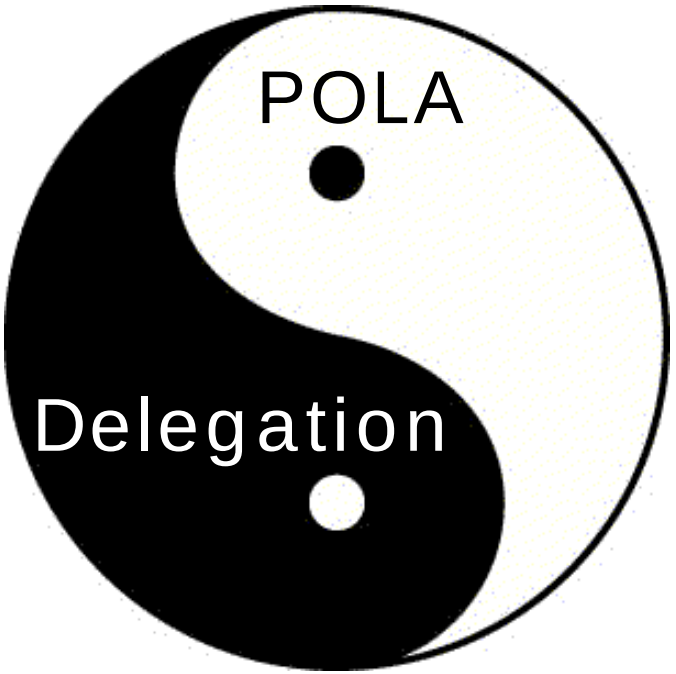
A PictureBook of Secure Cooperation



Marc Stiegler
Visiting Scholar, HP

Yin and Yang, POLA and Delegation



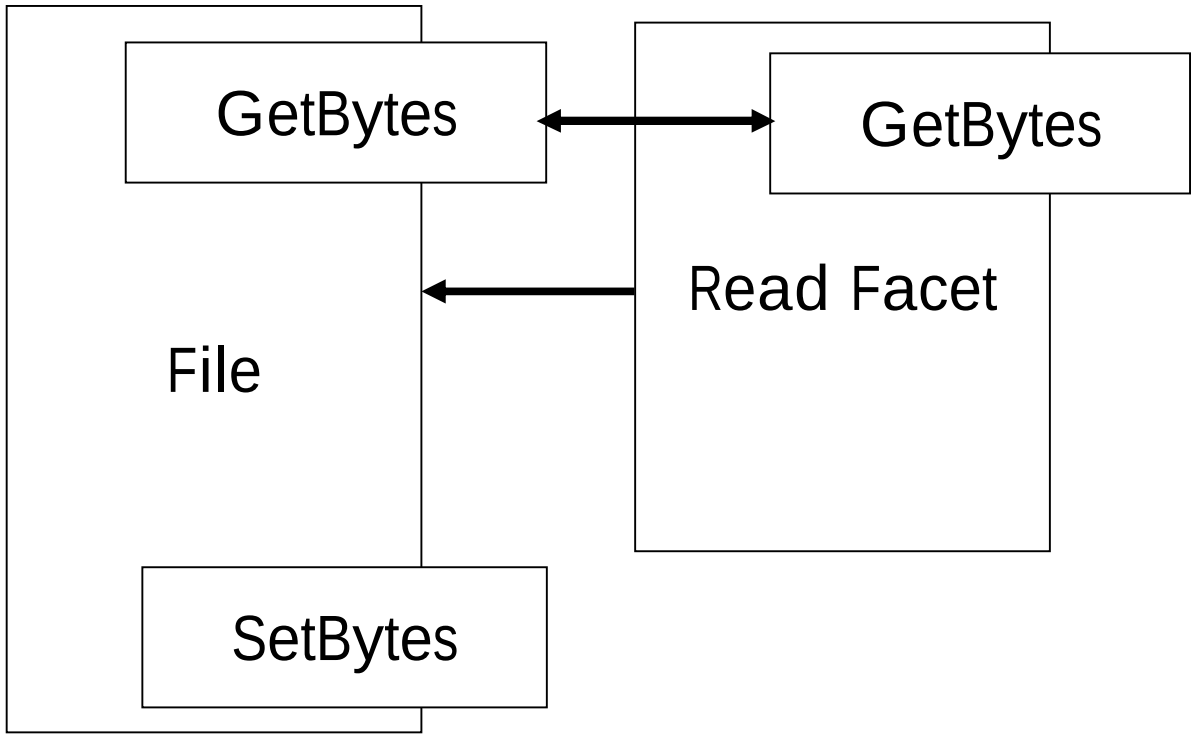
The Principle of Least Authority (POLA) simply states, give people exactly the authority they need, neither more nor less.

To have enough authority, one must often receive a part of someone else's authority: POLA requires delegation. To avoid excess authority, delegation must only include what is needed: delegation requires POLA.

POLA is required for security. Delegation is required for cooperation. Secure cooperation is not possible without both.

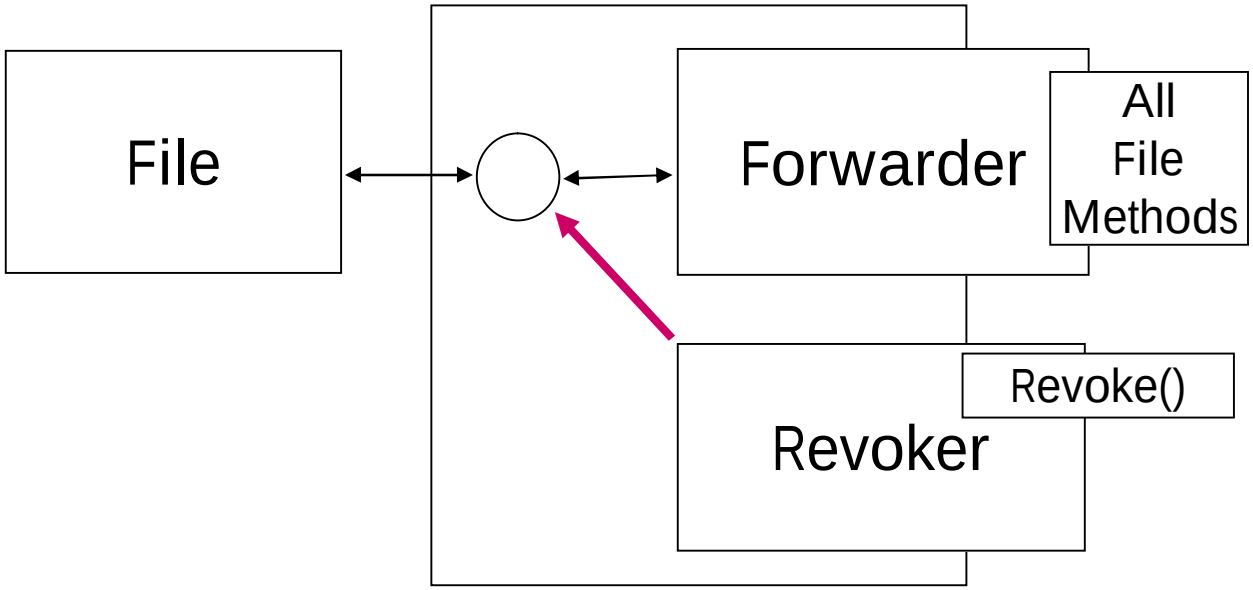
This picturebook presents a toolkit of composable building blocks from which an endless variety of patterns of secure cooperation may be built. Each element supplies both delegation and security, as do systems built with them

Facet



•A facet presents a subset of the authorities embodied in a more powerful object. In this example, a read facet is made for a simple read/write File object with 2 methods, getBytes() and setBytes(bytes). The getBytes method and its reply are transparently forwarded through the Read Facet; the setBytes method is not.

Revocable Forwarder

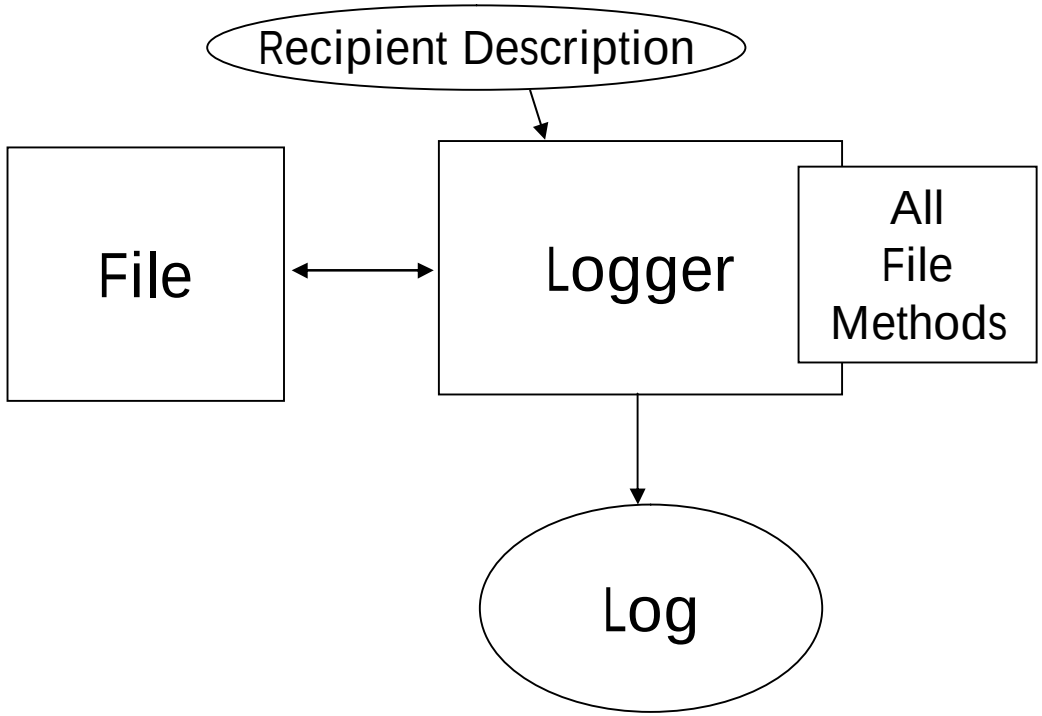


• A revocable forwarder has 2 parts: the forwarder simply forwards messages&replies to&from the underlying powerful object. The revoker, when called into action, destroys the link to the underlying object, effectively revoking the authority conveyed by the forwarder. The Revoker holds only the power to revoke; it can be given to people who are not trusted with the forwarded file authority.

Composition

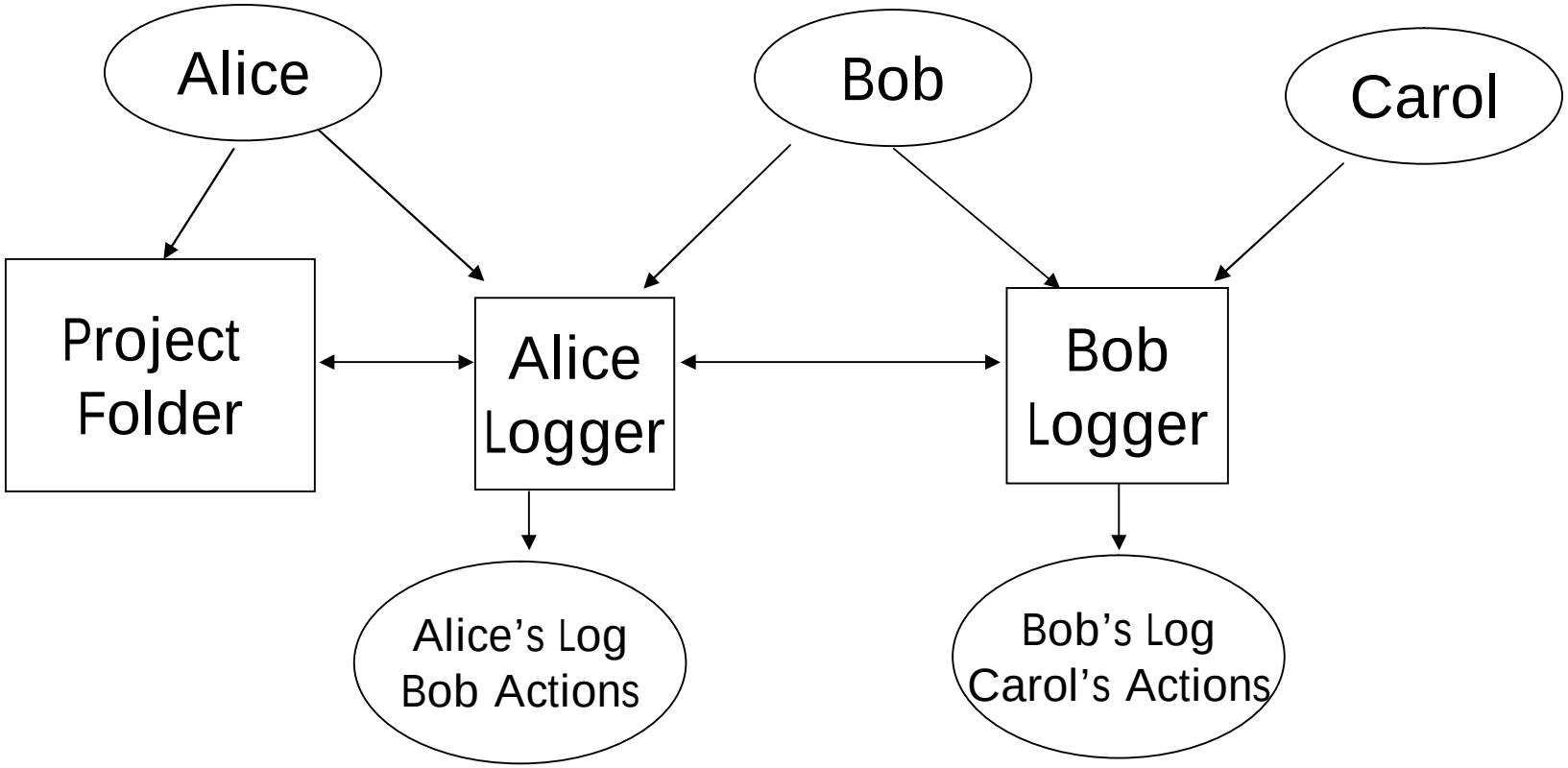
Already we have enough parts to do interesting compositions. Combining a read facet and a revocable forwarder, we create a temporary read authority. In practice, if a programmer needs to create a revocable read facet, he may create a single object rather than 2 as depicted here. But one can imagine tools, even graphical tools given to end users, that would allow tinker-toy construction of POLA-rized delegatable authorities as depicted here.

Logger (Logging Forwarder)



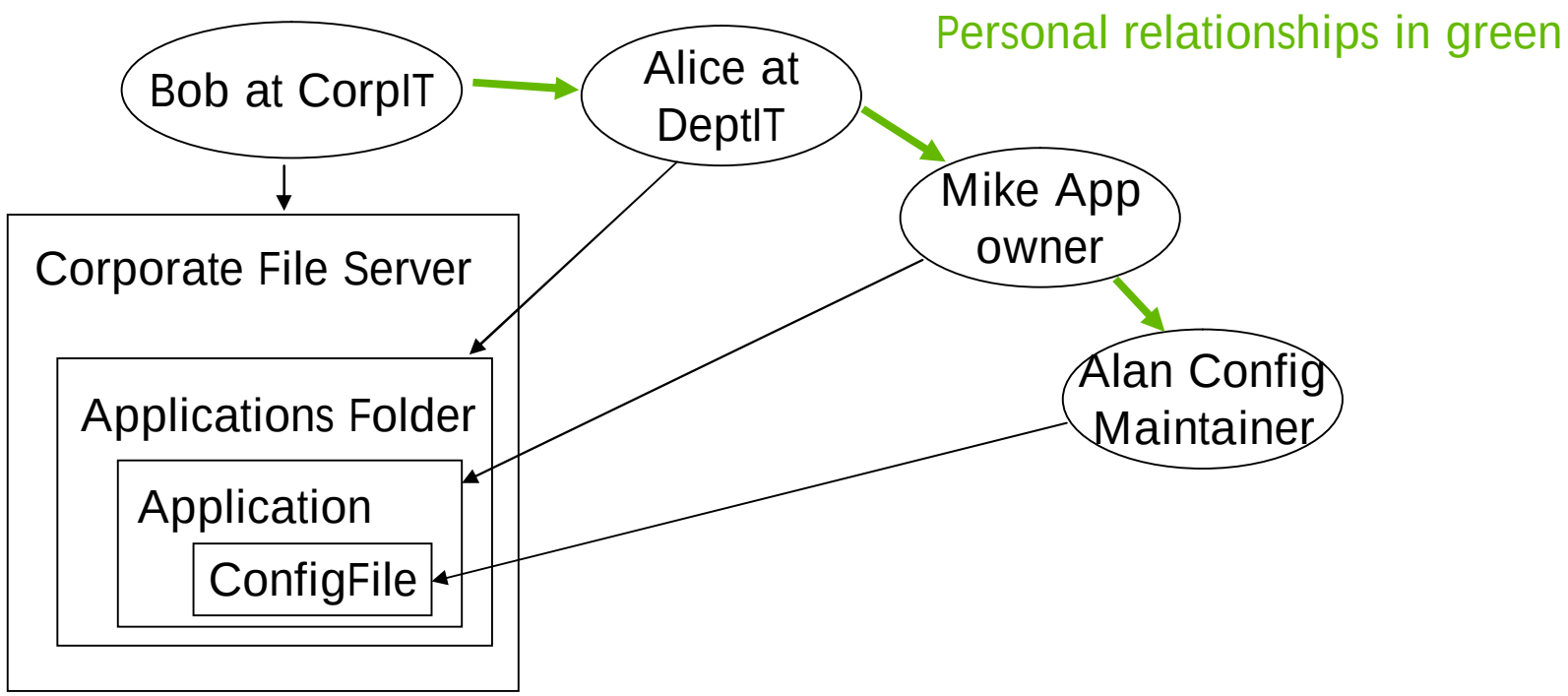
The logger is another forwarder that simply creates an audit trail of messages sent to the powerful authority by the recipient of the logger. The recipient description may be the “petname” of the entity to whom the authority is being granted. Petnames are described later.

Accountability



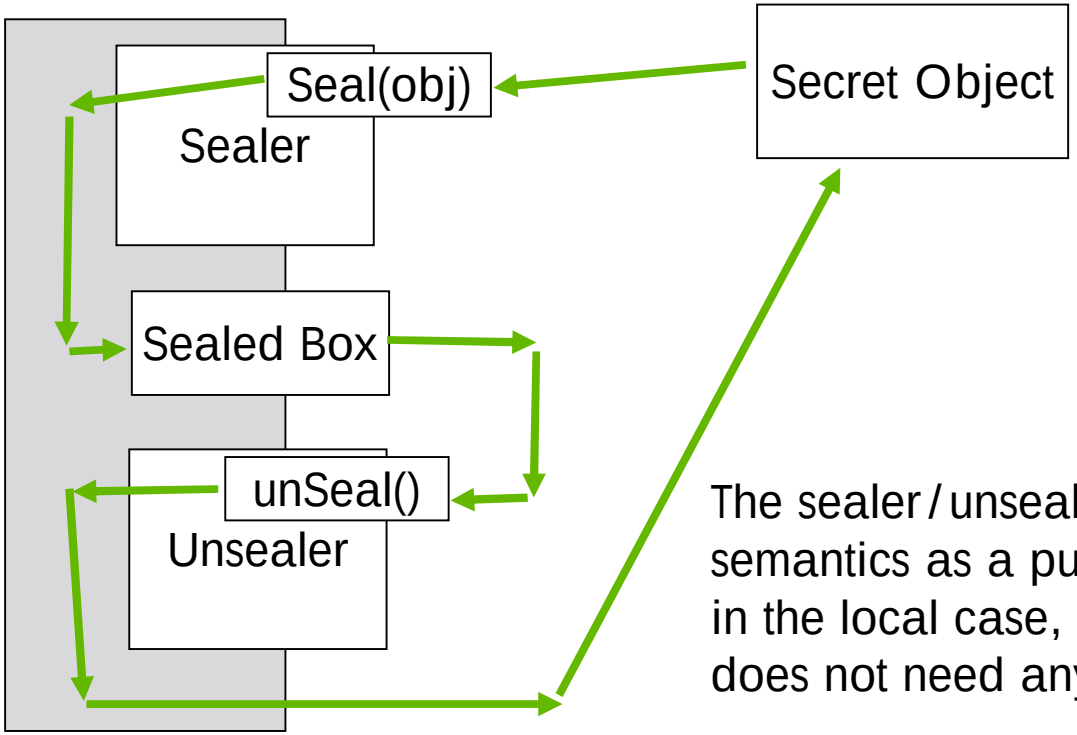
Alice is logging all Bob's actions on the project. She holds Bob accountable for anything that happens to the project via the logging forwarder she gave him. Bob holds Carol accountable for any action via the logger he gives to her. Alice holds Bob responsible for anything Carol does. This exactly parallels the way delegation and accountability occur in the physical world

Polarized Delegation Chain



A chain of delegations, each of which is more restricted. Alan maintains the configuration file for an application in the app folder on a corporate server. No person should have to grant an excess authority to anyone; no person should have to grant any authority to someone they do not personally have reason to trust. The composition should still work correctly even if Mike is an employee of the application vendor. If Alice revokes Mike, the Alan authority needs to be automatically revoked for security to be preserved in an intuitive, locally-comprehensible fashion. This is a real life example, only the names are changed.

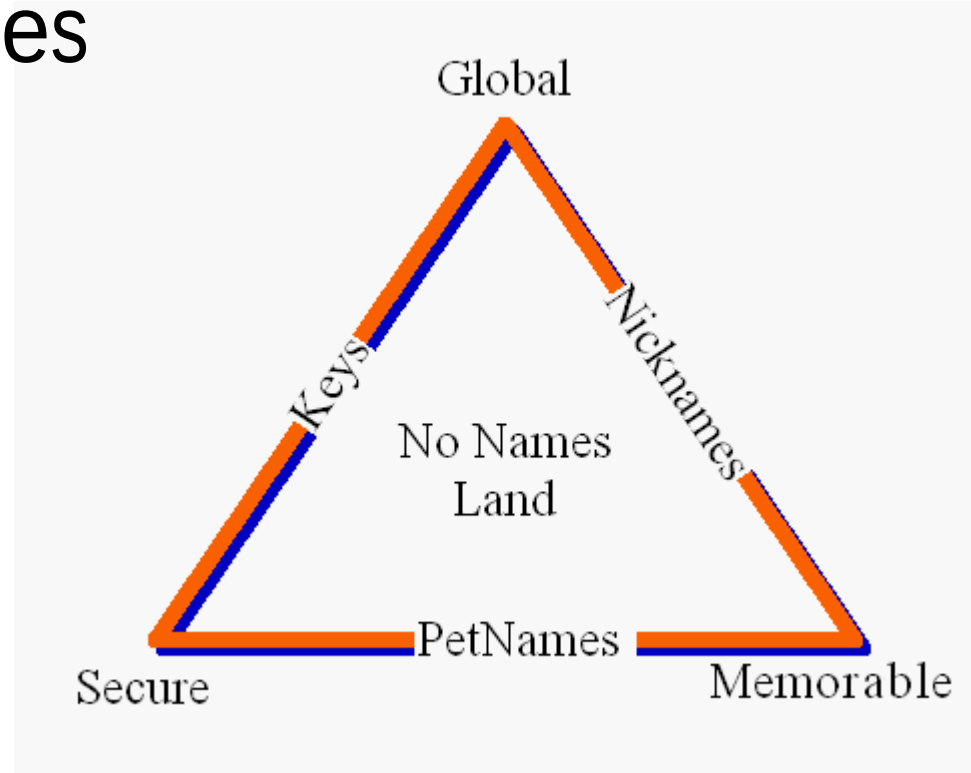
Sealer / Unsealer



The sealer / unsealer pair supplies the same semantics as a public / private key pair, but in the local case, using object references, does not need any actual crypto.

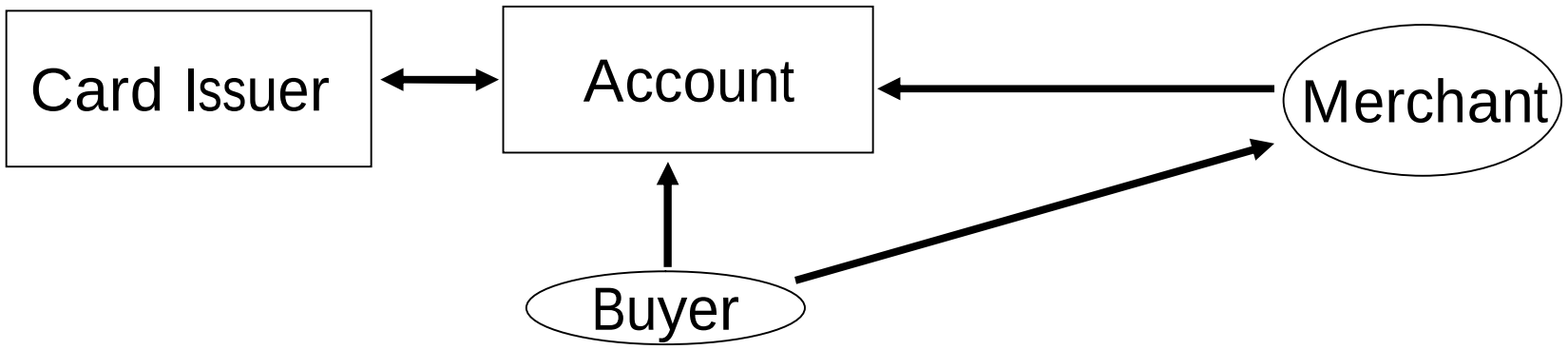
An object is placed in a sealed box using the sealer. The object can be retrieved from the box only by using the unsealer. If the sealer is made public, and the unsealer is held privately, anyone can send the holder of the unsealer a secret object (“encryption”). If the unsealer is made public, and the sealer held privately, anyone can know that the object was put in the box by the owner of the sealer (“signing”)

Petnames



Petname systems are able to surmount the paradox of Zooko's Triangle, which states that any single name can have only two of the three desirable properties: globality, secure uniqueness, and memorableness. Domain names, for example, are global and memorable, but are not securely unique (both forgeable and phishable). In a petname system, the key is global and secure. The nickname is the key owner's own suggestion as to the memorable name for himself, global and memorable. The petname is assigned by a user of the key; it is secure and memorable, but private rather than global.

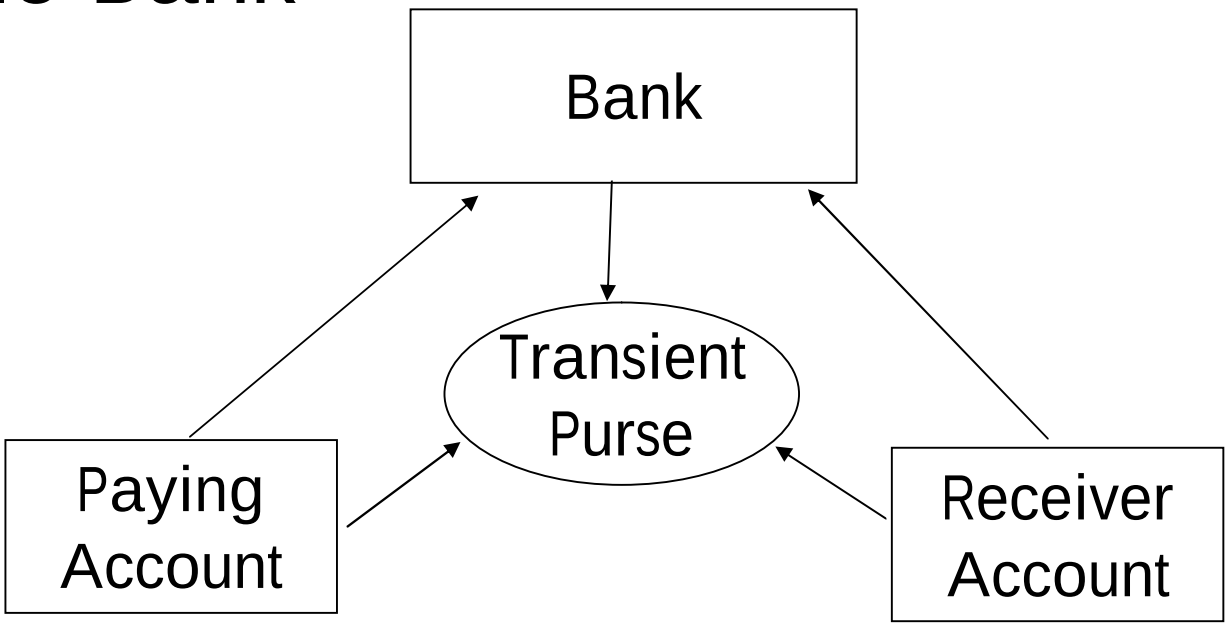
Finance 1: Credit Card Breach



In the credit card model of financial transactions, the owner of the card grants full authority over the whole credit line to the merchant; the merchant takes as much as he desires. The result is inevitable: fraud on a massive scale, resulting in risk management transaction overheads comparable to imposition of a second California State Sales Tax.

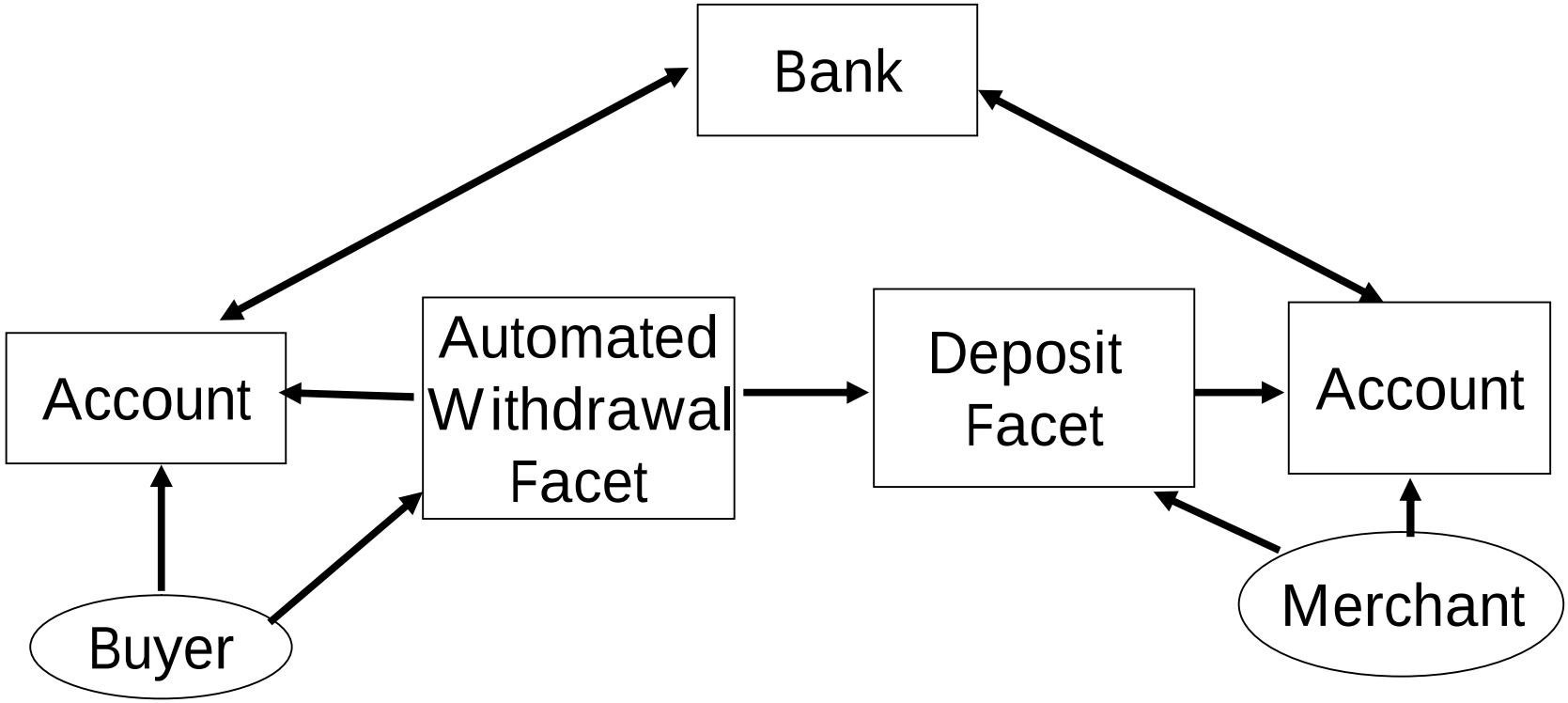
This is a pattern of failure, in which delegation is so important that people will delegate even without POLA, voluntarily breaching their own security. The consequences are both obvious (fraud, high overhead rates) and subtle (harsh restrictions on who may qualify as a merchant).

Simple Bank



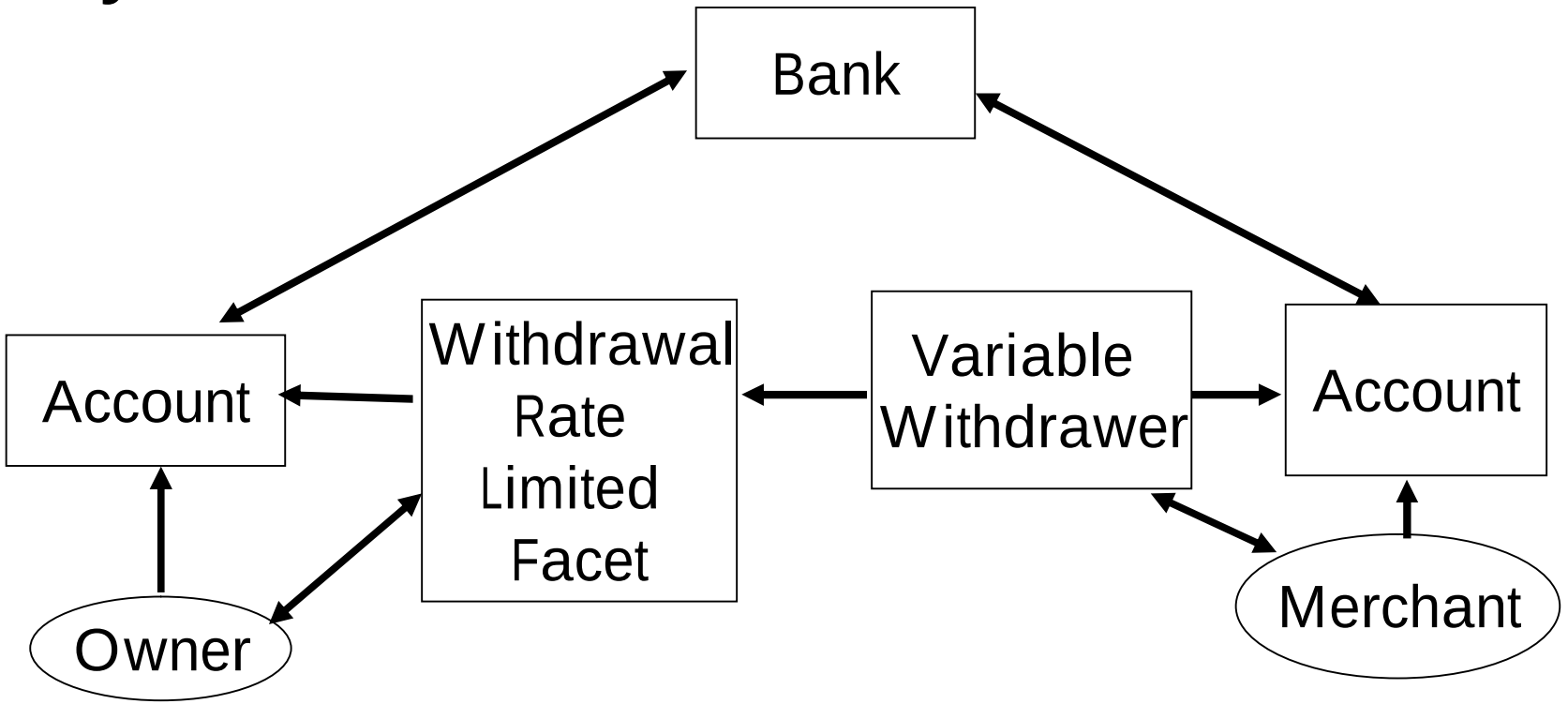
In a simple secure bank transfer, the payer creates a temporary purse to hold the payment, and gives the receiver a reference to the purse. The receiver is considered paid after bank acknowledgement that the money has been transferred from the purse to the recipient account. In the simplest version of this, the accounts and purses all use the same protocol, i.e., they are all purses. In the industrial version (the Waterken IOU protocol), an additional separate object, the transfer object, is introduced to accommodate more complex relationships.

Finance 2: Periodic Payment



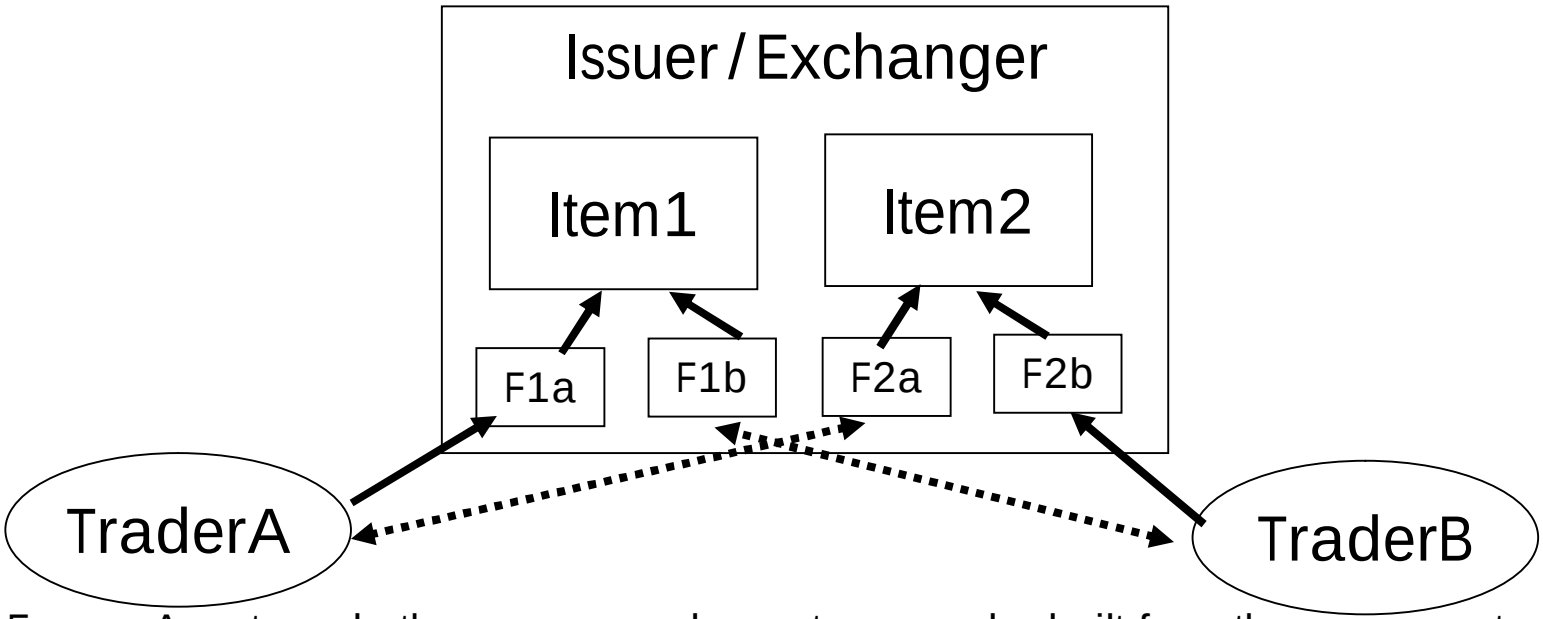
In the Periodic Payment, the merchant gives the buyer a deposit-only facet on his account. The buyer gives that facet to an object he creates with a reference to his own account, that periodically withdraws money from the account and sends it to the deposit facet. This is similar to the way the program Quicken automates monthly bills.

Finance 3: Variable Periodic Payment



A monthly electric bill fluctuates in cost. A risk-limiting approach for the buyer is to create a withdrawal-limited facet on his account for the vendor. The facet might implement, “allow withdrawals up to $350 every 30 days”. This facet is handed to the merchant, who hands it to the program that computes the monthly cost, and withdraws that amount. If the amount requested for withdrawal exceeds the authorization, both owner and merchant are notified.

Simple Escrow Agent



Escrow Agents and other more complex systems can be built from the components described earlier. Here we have an issuer of a pair of tradable items. The current owners of the items (i.e., the entities who have exclusive, though revocable, forwarders to the items) wish to trade. The owners place their items in an exchanger. Each trader is given a limited facet on the other owner's object used to verify the desired item. When both traders confirm the trade, each trader's original forwarder is revoked, and the verification facets are replaced / upgraded to full power exclusive references. The Diablo II item trading interface works this way. It demonstrates such exchanges can be so intuitive, no instructions are needed to operate safely and effectively.

The Missing Pictures

The patterns described in this picturebook are simple because they discard the modern fascination with the identities of the participants. Individual Authentication is so pervasive, it is now a part of the problem. Suppose that your car, instead of accepting a delegatable key, demanded that your driver's license match the car's title registry, which happens to be in your spouse's name. Entrepreneurs would leap forward to develop ever more powerful "identity management" for automobiles. We would subcontract to security experts so our teenage daughters could borrow the car to buy milk. Heaven forfend that the daughter, breaking her leg, had to delegate to her best friend to get to the hospital.

The patterns here do not require individual authentication. Rather, these patterns focus on authorization: ask not, "who are you?", but rather, "are you allowed?". This was always the crucial question anyway; by asking this better question, we get a better answer.

Many of the latest buzzwords and hottest technologies in computer security answer the wrong questions. They often actively hinder delegation or POLA or both. A world of secure cooperation would dispense with them. No one would miss their passing. Offenders include:

Multiple Passwords. Centralized Identity Management. Federated Identity Management. Access Control Lists. True Name authentication. Certificate Authorities. File Attachment Suppressors. Trust Zones. Firewalls.

These systems are absent from this picturebook. If this disturbs you, feel welcome to contact the author for more details about their true relevance in a clean, well-lighted place for secure cooperation.

Sample Code in E

Following are samples of code for secure cooperation written in E. For more information about E, and secure cooperation in general, please visit

http://www.erights.org

Facet

```
def makeReadOnlyFile(fullPowerFile) {
    def readOnlyFile {
        to getBytes() { return fullPowerFile.getBytes() }
    }
    return readOnlyFile
}
```

Revocable Forwarder

```
def makeForwarderRevokerPair(var target) {
    def forwarder {
        match [verb, args] { E.call(target, verb, args) }
    }
    def revoker {
        to revoke() { target := null }
    }
    return [forwarder, revoker]
}
```

Logger (Logging Forwarder)

```
def makeLogger(accessedObject, recipientName, logWriter) {
    def logger {
        match [verb, args] {
            logWriter.println("Access: " + recipientName)
            logWriter.println("Method: " + verb)
            for each in args { logWriter.println("    with arg: " + each) }
            E.call(accessedObject, verb, args)
        }
    }
    return logger
}
```

Sealer / Unsealer

```
def makeSealerUnsealerPair() {
    var shared := def none {}
    def sealer  {
        to seal(obj) {
            return def box { to share() { shared := obj }}
        }
    }
    def unsealer {
        to unseal(box) {
            shared := none
            try { box.share(); require(shared != none)
                  return shared
            } finally { shared := none }
         }
    }
    return [sealer,unsealer]
}
```

Simple Bank

```
def makeBank(name :String) :any {
    def [sealer, unsealer] := makeBrandPair(name)
        return def bank {
        to makePurse(var balance :(int >= 0)) :any {
            def decr(amount :(0..balance)) :void {
                balance -= amount
            }
            return def purse {
                to getBalance() :int   { return balance }
                to makePurse() :any { return bank.makePurse(0) }
                to getDecr()       :any { return sealer.seal(decr) }
                to deposit(amount :int, src) :void {
                    unsealer.unseal(src.getDecr())(amount)
                    balance += amount
} } } } }
```

Acknowledgements

Thank you to all the members of the E-lang community, especially Fred Spiessens, Julien Couvreur, Chip Morningstar, Mark Miller, and Alan Karp, all of whose comments led to improvements in the current draft.

Also thank you to John Wilkes, who inspired the creation of this document.