Reacting to change
Key principles
Key principles

• Scenarios drive an interface design
Key principles

• Scenarios drive an interface design

• Attributes define an object interface
Key principles

• Scenarios drive an interface design

• Attributes define an object interface

• Notifications communicate change
Desirable side effects

• Posting notifications
• Reacting to notifications
• Notification delivery as an activity
N.B.: This is an undervalued topic

- Most coverage is superficial and generally wrong
  - Callbacks, listeners, observers

- We will get it right and cover the important stuff
  - Data model, control flow, transaction semantics
Object interactions
Object interactions

- Object A calls a mutator on Object B
- Object C is interested knowing about changes to B
- Object A and Object C interact with B independently
- Object B only knows some objects are interested in changes
Objects A and C don’t (can’t) know about each other

1. Object B’s mutator posts a notification AND

2a. Object C asks Object B for notifications OR

2b. A general event log records all Object B notifications and provides access to Object C (and other interested objects)
2a is lighter weight and more broadly applicable

- Object C expresses interest in B’s notifications
- Object A calls the mutator on B
- Object B posts a notification
- Object C receives the notification
Terminology

- Notifier — Object B
- Notifiee — Object C
- Notification — Description of what the mutator did
Counter example (HTML+JavaScript)

• Notifier — Counter

• Notifiee — CounterReactor

• Notification — onCount

• Demo
C++ Example: Bank account

class BankAccount : public NamedInterface {
public:

  ...  
  double balance() {
    return balance_;  
  }

  void balanceIs(const double balance) {
    balance_ = balance;  
  }

protected:

  double balance_;  
  ...  
};
Nested class for notifications

// Inside BankAccount class

class Notifiee : public ... {
public:

    virtual void onBalance() {
        // Default is to do nothing.
    }

};
Notification methods

• Our convention: “on” + attribute name

• Other conventions: attribute name + “Changed”
  e.g., “balanceChanged” (or more generally a verb using past tense)
Important semantics

• Notification methods MAY NOT return a value

• Notification methods MAY NOT throw an application exception

• This is a key difference between exceptions and general callbacks e.g., Java VetoableChangeListener

• Notifications allow deferred OR inline execution
Notifier object refers to notifyee(s)

// public inside BankAccount class

// singleton example
    Notifiee* notifiee_;

// list example
    std::list<Notifiee*> notifiees_;
Notifier mutator posts a notification

```cpp
void BankAccount::balanceIs(const double balance) {
    if (balance_ != balance) {
        balance_ = balance;

        // singleton case
        if (notifiee_) {
            notifiee_->onBalance();
        }
    }

    OR

    // list case
    for (const auto n : notifiees_) {
        n->onBalance();
    }
}
```
Mutator makes sure not to throw any exceptions

```cpp
void BankAccount::balanceIs(const double balance) {
    if (balance_ != balance) {
        balance_ = balance;

        for (const auto n : notifiees_) {
            try {
                n->onBalance();
            } catch (...) {
                // Must be a system exception, e.g., null pointer
                // Can use an extensible mechanism,
                // similar to logging
            }
        }
    }
}
```
Convenience function(s) to post a notification

```cpp
using fwk::NotifierLib::post;

void BankAccount::balanceIs(const double balance) {
    balance_ = balance;
    post(this, &Notifiee::onBalance);
}
```
Convenience function(s) to post a notification (cont’d)

template <class T>
void post(T* const notifier, void (T::Notifiee*func)()) {
    for (const auto n = notifier->notifiees()) {
        try {
            (n->*func)();
        } catch (...) {
            n->onNotificationException();
        }
    }
}
Deferred vs inline notifications

• Post operation could simply queue the notification

• Different notifiees could have different queues

• Delivery could be asynchronous with respect to the mutator
Application logic should *not* depend on whether notification delivery is deferred or inline.
Notification meaning

- Common case is absolute mutator (onBalance)
- Relative mutators might have different notifications
  Stocks: onShareSplit and onShareBuyBack
Collection mutators pass selector in notification

class Directory : public NamedInterface {
public:
  ...
  Ptr<Directory> subdirNew(const string& name) {
    const Ptr<Directory> subdir = ...;
    post(this, &Notifiee::onSubdirNew, name);
    return subdir;
  }
  ...
}
Notification parameters

• Can additionally pass new and/or old values
  • Previous and current balance
  • Insert/delete/replace (multiple) characters in text

• Beware: Parameter values might not be current
  • Mouse/touch position
  • Should not require notification delivery before further mutator calls
Implement reactions by subclassing Notifiee

class AccountReactor : public Account::Notifiee {
public:

    static Ptr<AccountReactor> instanceNew(const Ptr<Account>& a) {
        const Ptr<AccountReactor> r = new AccountReactor();
        r->notifierIs(a);
        return a;
    }

    void onBalance() {
        cout << "new balance is " << notifier()->balance() << endl;
    }

};
Need a little help from the Notifiee classes

class Account : public NamedInterface {
public:
    ...
class Notifiee : public BaseNotifiee<Account> {
public:

    void notifierIs(const Ptr<Account>& account) {
        connect(account, this);
    }

    virtual void onBalance() { }
};
    ...
};
Putting everything together

```cpp
const auto account = Account::instanceNew("MyAccount");
const auto reactor = AccountReactor::instanceNew(account);

account->balanceIs(100);
// Output is “new balance is 100”

account->balanceIs(200);
// Output is “new balance is 100”

account->balanceIs(200);
// No additional output
Example with collections: Track parking spaces

• ParkingLot has a collection of PackingSpace objects, HandicappedSpace is a subclass of Space

• Sensors detect when a car arrives or leaves, calling space->occupiedIs(true or false)

• Spaces are numbered 100, 101, 200, 201, etc., where the first digit is the floor

• We want to track the number of available non-handicapped spaces on each floor
Parking space notifications

- ParkingLot
  - onSpaceNew(space) when a space is added to the lot
  - onSpaceDel(space) when a space is removed from the lot

- ParkingSpace
  - onOccupied() when a car arrives or leaves
Tracking individual spaces

class SpaceTracker : public ParkingSpace::Notifiee {
private:

    friend class LotTracker;

    Ptr<LotTracker> lotTracker;

    void onOccupied() {
        // “trampoline” bounce
        lotTracker->onSpaceOccupied(notifier());
    }
};
Tracker for the lot

class LotTracker : public ParkingLot::Notifiée {
public:

    void onSpaceNew(const Ptr<ParkingSpace>& space) {
        if (dynamic_cast<HandicappedSpace*>(space.ptr()) == null) {
            addSpace(space);
            if (!space->occupied()) {
                available[floor(space->number())] += 1;
            }
        }
    }

...
Setting up the trampoline

private:

    . . .

    void addSpace(const Ptr<ParkingSpace>& space) {
        const Ptr<SpaceTracker> tracker = new SpaceTracker();
        tracker->lotTracker = this;
        tracker->notifierIs(space);
        spaceTracker[number] = tracker;
    }

    . . .
Reacting to the occupied change

```cpp
void onSpaceOccupied(const Ptr<ParkingSpace>& space) {
    const int delta = space.occupied() ? -1 : 1;
    available[floor(space->number())] += delta;
}
```
Cleaning up

... void onSpaceDel(const Ptr<ParkingSpace>& space) {
    const auto iter = spaceMap.find(space->number());
    if (iter != spaceMap.end()) {
        spaceMap.erase(iter);
    }
} ...

C++ Sidebar: BaseNotifiee makes the connection

template <class Notifier>
class BaseNotifiee : public RootNotifiee {
public:

    const Ptr<Notifier>& notifier() {
        return notifier_;  
    }

protected:

    Ptr<Notifier> notifier_;
BaseNotifiee makes the connection (cont’d)

```java
BaseNotifiee() {
    // Nothing to do.
}

~BaseNotifiee() {
    disconnect();
    notifier_ = null;
}
```
void connect(
    const Ptr<Notifier>& notifier,
    typename Notifier::Notifiee* const notifiee
) {
    if (notifier_ != notifier) {
        if (notifier_ != null) {
            disconnect();
        }
        notifier_ = notifier;
        if (notifier != null) {
            notifier->notifiees().push_back(notifiee);
        }
    }
}
BaseNotifiee disconnecting

```cpp
void disconnect() {
    if (notifier_ != null) {
        auto& list = notifier_--notifiees();
        for (auto i = list(); i != list(); ++i) {
            if (*i == this) {
                list.erase(i);
                break;
            }
        }
    }
}
```

```cpp
};
```
Summary so far: Notifications provide logical structure

- One-way communication from notifier to zero or more notifiees
  - No return value
  - No exceptions
- Part of the object interface
- Deferred execution simplifies control flow
- In contrast, callbacks are low-level mechanism where an arbitrary callee is passed as a parameter