Design Patterns in the Wild

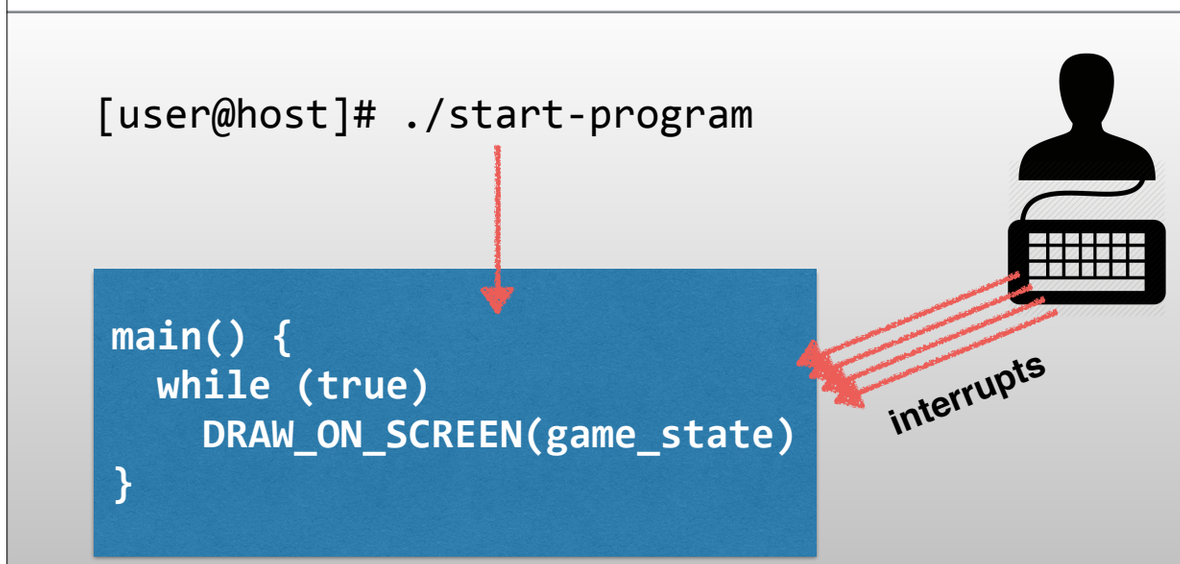
Events
Dispatchers
MVC
Frameworks
Dependencies
Delegates
Architecture

Concepts

- Event-driven programming
- Model-view-controller
- Library vs. framework
- Dependency management
- Delegation Patterns

User input as interrupts

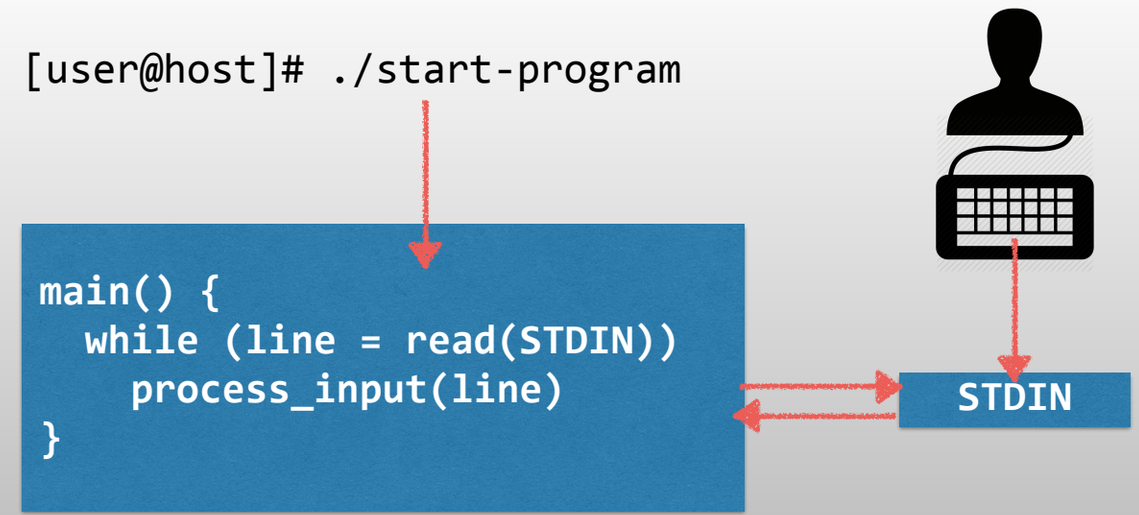
Writing interruptible programs is hard!



Why do it this way?

User input as a file

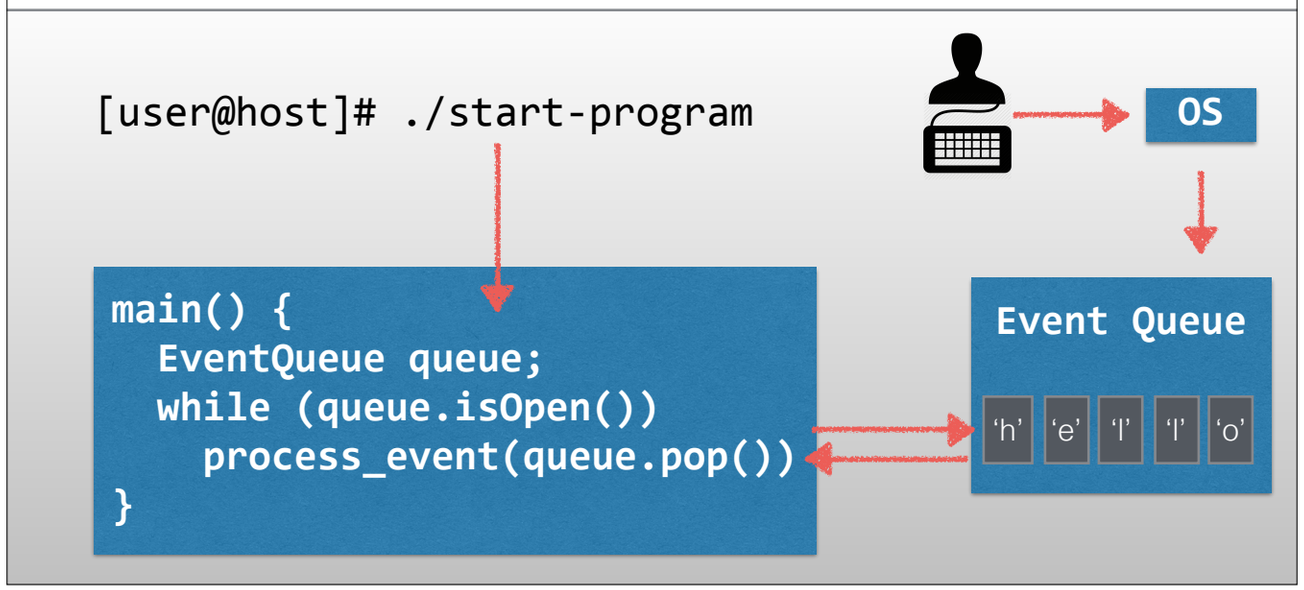
To the program, you are a file to be read.



Why do it this way?

User input as events

Usually refers to *event queue*-driven programs.



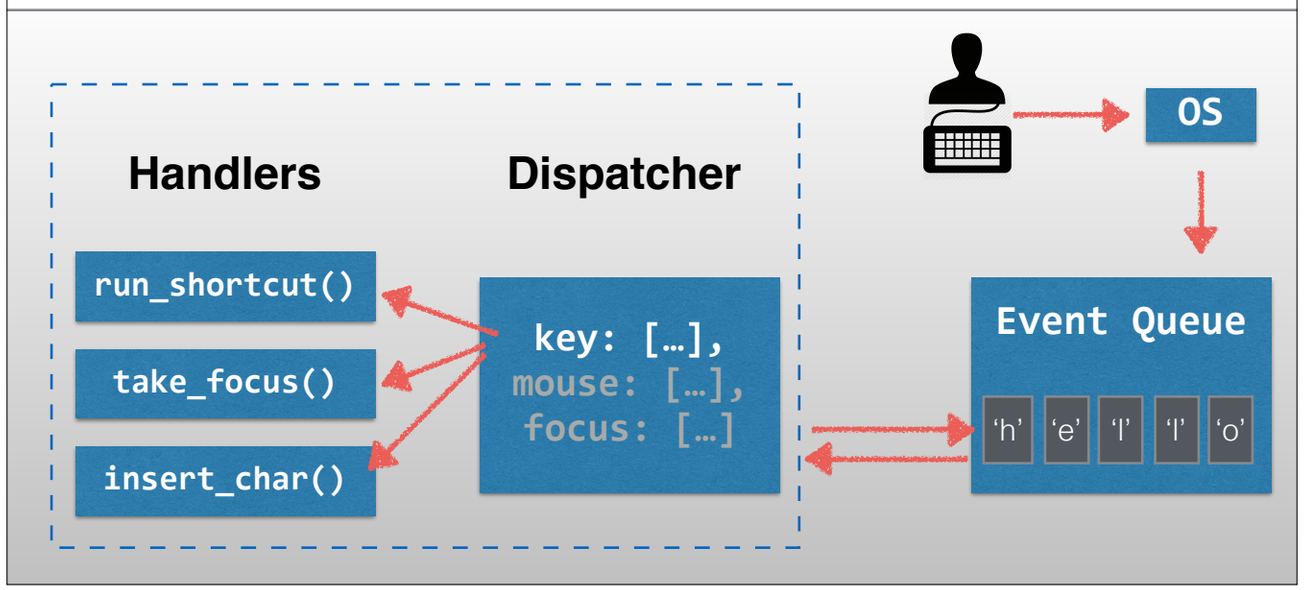
Event queue: provided by OS or framework; serializes event order.

How does the OS decide what app's event queue to use?

What does process_event do?

Event-driven programming

Usually refers to *event queue*-driven programs.



How does the OS decide what event queue to use?

Many different models for event dispatching (iOS responders, DOM events, …)

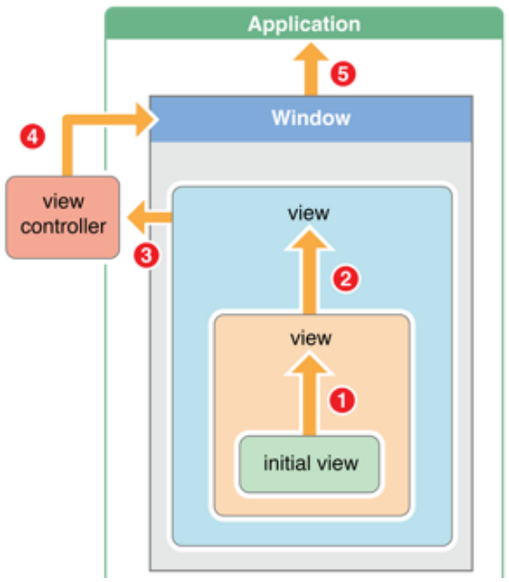
Used for GUI and non-GUI applications

Event dispatching (Cocoa)

Two-phase dispatch:

0. hit-test for NSView
1. try first responder
2. try hit-tested view, then go up responder chain

Supports *pre-emption* by first responder and *fallback handling* by responder chain

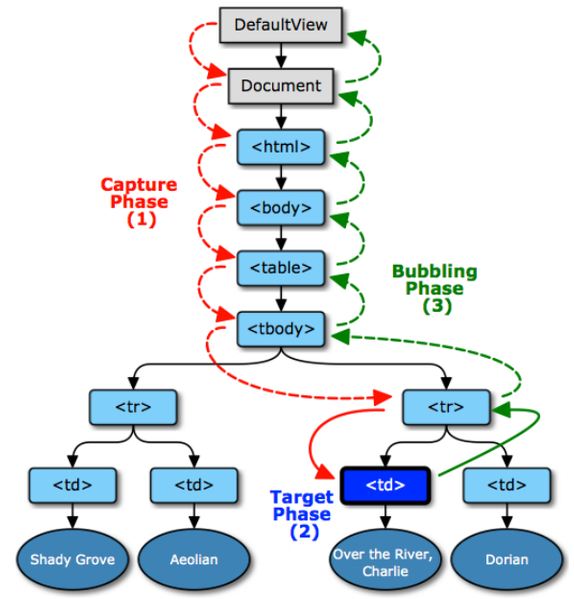


Event dispatching (DOM)

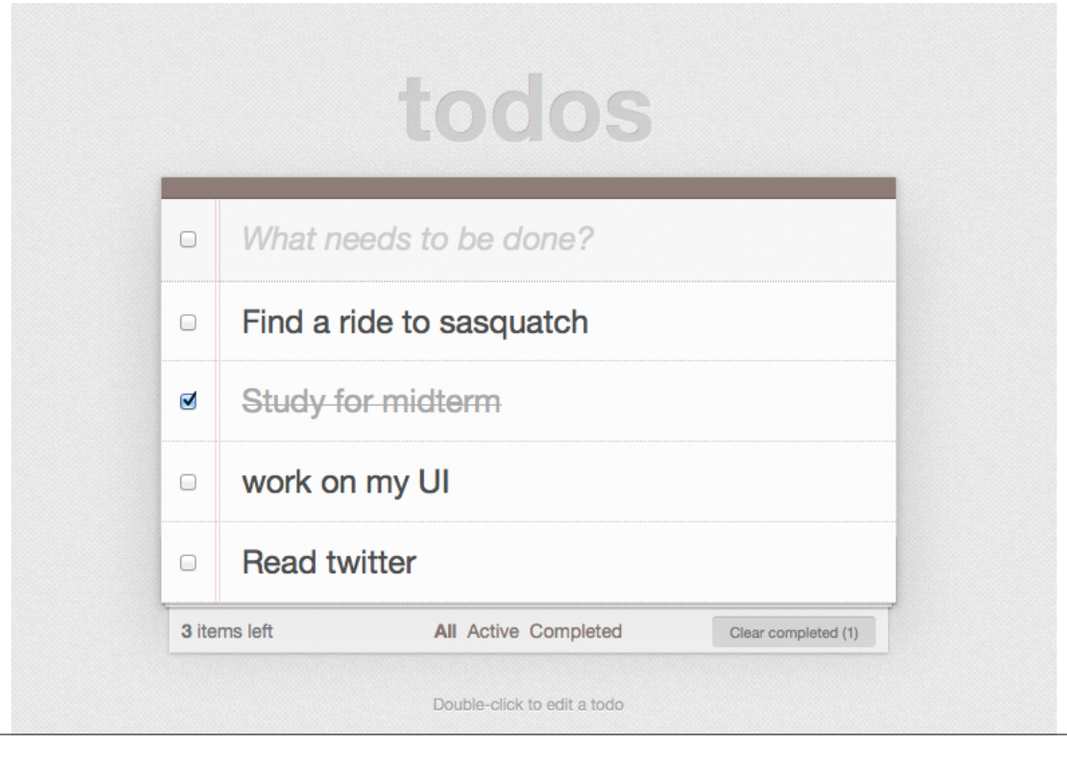
Three-phase dispatch:

0. hit-test for DOM node
1. from root to target
2. hit the target
3. from target to root

Supports *pre-emption* and *fallback handling* by parent elements



GUI Architecture



How would you structure the code?

Classes:

TextInput

Checkbox

Button

Filter?

TodoList?

TodoItem?

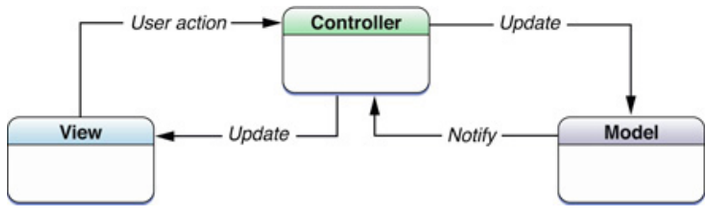
TodoApp?

http://todomvc.com/architecture-examples/emberjs/

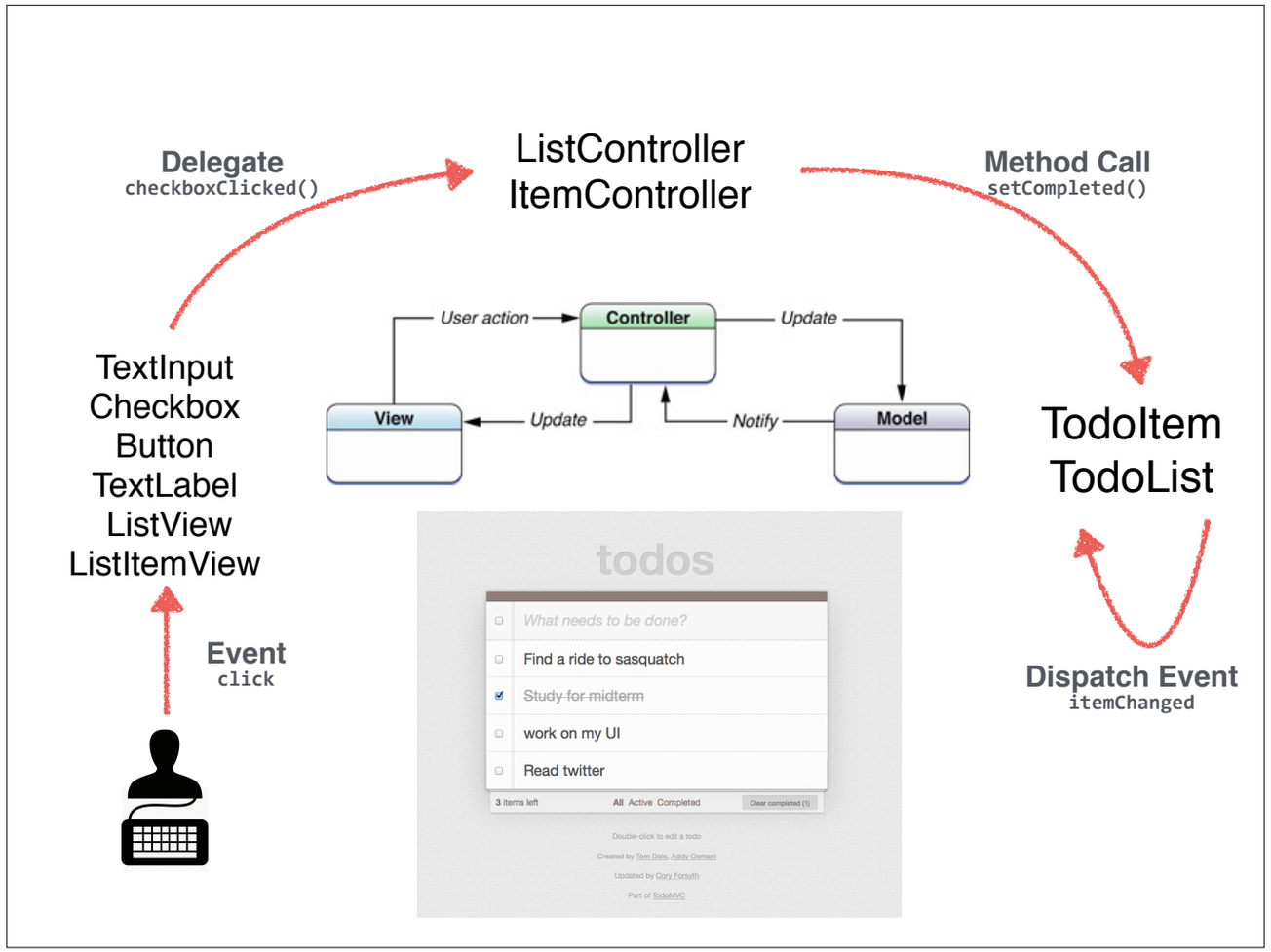
How do the classes relate to each other?

Model-View-Controller

Every class is a *Model*, *View*, or *Controller*.



- **Model**: stores data and manages inter-object relationships
- **View**: what's displayed; how the user sees and interacts
- **Controller**: interprets user actions, updates models and views

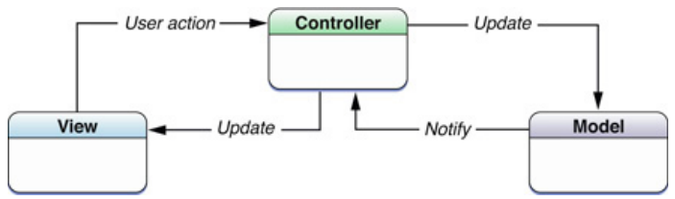

Delegate
checkboxClicked()
ListController
ItemController
Method Call
setCompleted()
User action
Controller
Update
View
Update
Notify
Model
TextInput
Checkbox
Button
TextLabel
ListView
ListItemView
TodoItem
TodoList
todos
What needs to be done?
Find a ride to sasquatch
Study for midterm
work on my UI
Read twitter
3 items left
All Active Completed
Clear completed (1)
Double-click to edit a todo
Created by Tom Dale, Addy Osmani
Updated by Cory Forsyth
Part of TodoMVC
Event
click
Dispatch Event
itemChanged

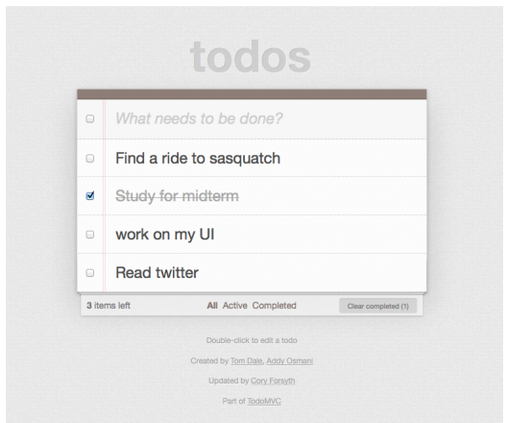
Method Call
`setChecked(true)`
`addClass('.done')`

ListController
ItemController

Event
`itemChanged`

TextInput
Checkbox
Button
TextLabel
ListView
ListItemView





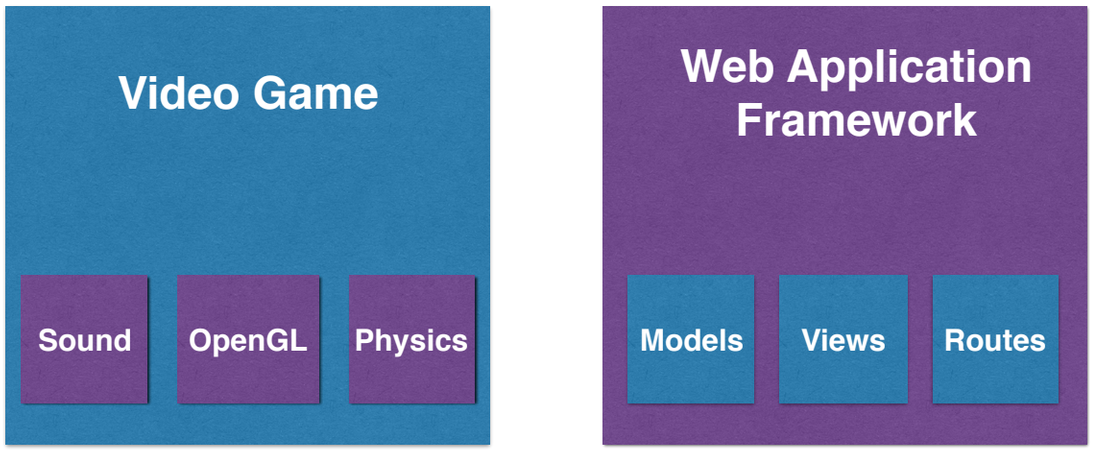
TodoItem
TodoList

Dispatch Event
`itemChanged`

MVC variations

- “View controller”: owns and manages views
  (back-forward history, filter/scroll state, focusing)
- “Model controller”: owns and manages models
  (undo/redo, import/export/create)
- Data binding: 1-way and 2-way model/view update
  (less boilerplate wiring of models and views)

Library vs. Framework



Discuss with people around you: what frameworks and libraries are you using? what has been helpful and what has been a pain?

Library vs. Framework

Goals of libraries:
selective code reuse,
specialize in a few capabilities,
maximum versatility,
backwards compatibility

Drawbacks of libraries:
requires more architecture,
good documentation is rare,
partially solves problems

Goals of frameworks:
code and architecture reuse,
minimize boilerplate code,
maximize productivity,
community knowledge

Drawbacks of frameworks:
architecture lock-in,
magical minimal code,
steep learning curve,
inherent complexity

Satisfying Dependencies

Direct instantiation
A component can create the dependency, typically using the constructor of a hardcoded class name.

Service locators
A component can look up the dependency in a global, singleton registry of components.

Dependency Injection
The component can have the dependency passed to it where it is needed (by constructor argument, or setter)

Dependency Examples

Direct Instantiation

```
TextEditor::TextEditor() {
    m_linebreaker = new EnglishLinebreaker();
}
```

Service Locator

```
TextEditor::TextEditor() {
    m_linebreaker = Registry::instance()
                        .lookupComponent(Linebreaker::InterfaceID());
}
```

Dependency Injection

```
// Caller figures out which instance to supply.
TextEditor::TextEditor(Linebreaker* linebreaker) {
    m_linebreaker = linebreaker;
}
```

The details of how instances are supplied varies widely.

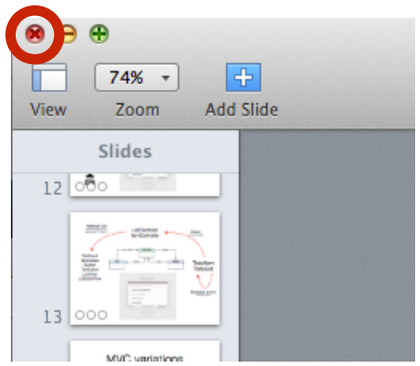
Generally its handled by configuration files and a loader/DI framework.

Why dependency injection?

Loose coupling.

- It makes it easier to test components separately.
- Strictly enforces “programming to the interface”.
- Avoids “abstract factory” and similar patterns.
- Supports lazy loading of specific implementations.

Delegation patterns



Who decides what clicking that button does?

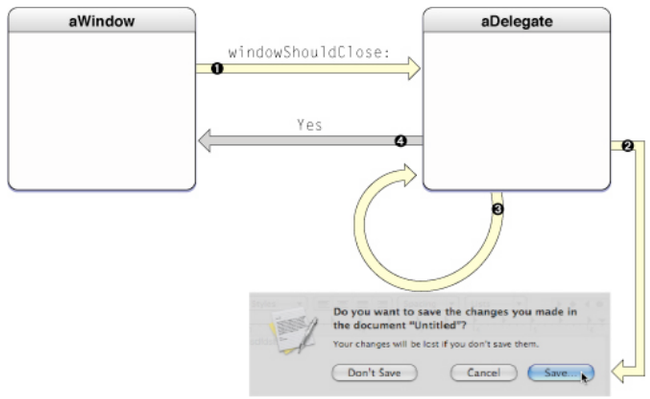
What if I haven't saved my work?

Delegation patterns

Subclassing
Statically decide who delegates what, using abstract methods on a base class or interface.

Delegates
Statically enforce interface contract, but dynamically swap out who provides it.

Events
Dynamically decide who responds to a message.



Architecture in the Wild

aosabook.org

Self-describing data types

COM, XPCOM, and the like

```
Interface IUnknown {
    abstract bool QueryInterface(classid);
    void addRef();
    void removeRef();
}

void makeRequest(IUnknown o) {
    if (!o.QueryInterface(INetwork))
        return;
    o.addRef();
    network = (INetwork)o;
    network.doStuff();
    o.removeRef();
}
```

- Components do not assume anything about components.
- Everything is manually reference-counted.
- Interfaces support language-independent interoperability.
- Analogs in other ecosystems (ObjC, Ruby, Python)