It’s Alive!

Continuous Feedback in UI Programming

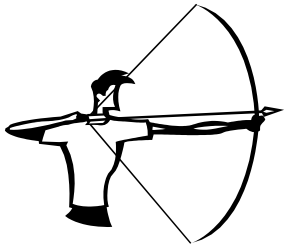
Sebastian Burckhardt
Manuel Fahndrich
Peli de Halleux
Sean McDirmid
Michal Moskal
Nikolai Tillmann
Microsoft Research

Jun Kato
The University of Tokyo

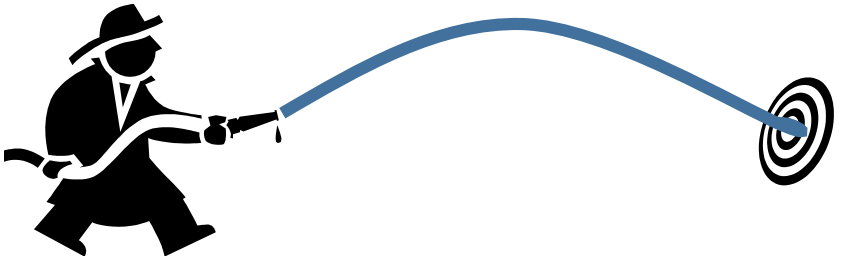
Live Programming : Archer Analogy

[Hancock, 2003]

- Archer:
  aim, shoot, inspect, repeat



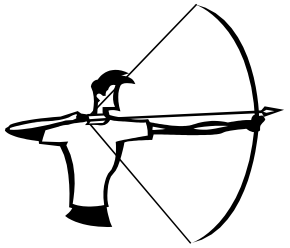
- Hose:
  aim & watch



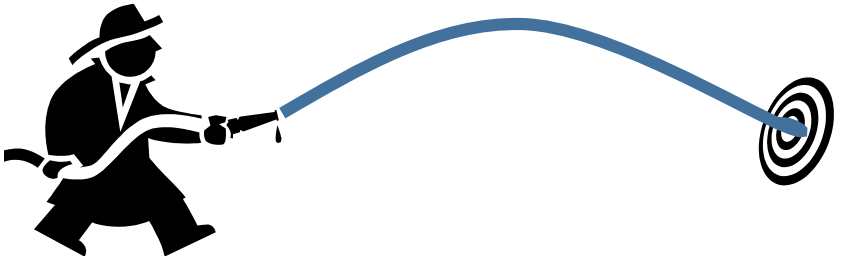
Live Programming : Archer Analogy

[Hancock, 2003]

- Archer:
  aim, shoot, inspect, repeat
- edit, compile, test, repeat



- Hose:
  aim & watch
- edit & watch



Quick Demo:

What is Live Programming?

What is TouchDevelop?

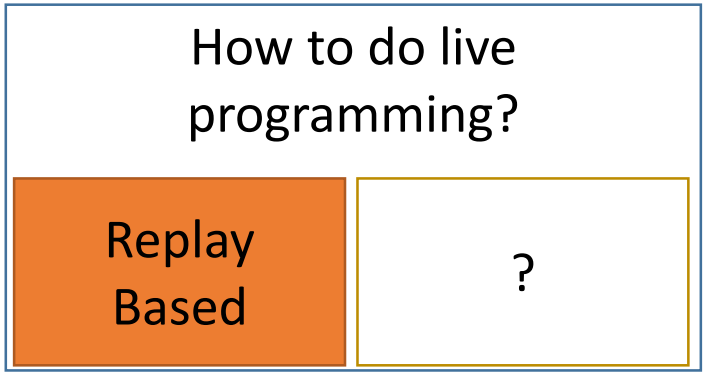
Question:
How to do live programming?

- Target:
  Event-driven apps with graphical user interfaces (GUI's)
  - User input events (tap button, edit text, ..)
  - I/O events (e.g. asynchronous web requests)

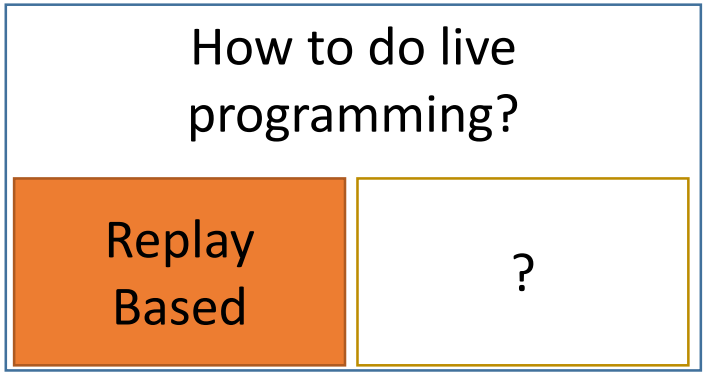
- We can think of code editing as an event
  (replace old program with a new one)

- What should we do in this situation?

on code changes,
just replay execution from beginning

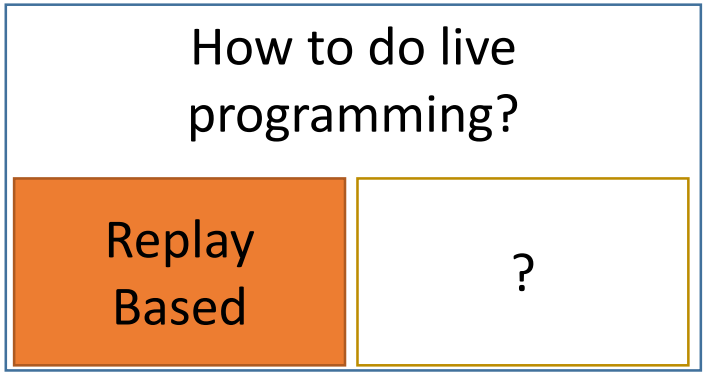


on code changes,
just replay execution from beginning



- Inputs?
  - Must record or repeat user inputs and I/O
- Divergence?
  - Recorded events may no longer make sense after code change
- Side effects?
  - Replaying external side effects can have surprising consequences
- Performance?
  - Apps with GUIs can run for a long time, replay not efficient

on code changes,
just replay execution from beginning



Replay is difficult. Worse: it does not always make sense.

- Inputs?
  - Must record or repeat user inputs and I/O
- Divergence?
  - Recorded events may no longer make sense after code change
- Side effects?
  - Replaying external side effects can have surprising consequences
- Performance?
  - Apps with GUIs can run for a long time, replay not efficient

Widen the Scope.

Question:
How to do live programming?

~~Replay Based~~

?

Widen the Scope.

Question 1:
How to program event-driven apps with GUIs?

- Retained Model-View
- Stateless Model-View

Question 2:
How to do live programming?

- ~~Replay Based~~
- ?

Widen the Scope.

Question 1:
How to program event-driven apps with GUIs?

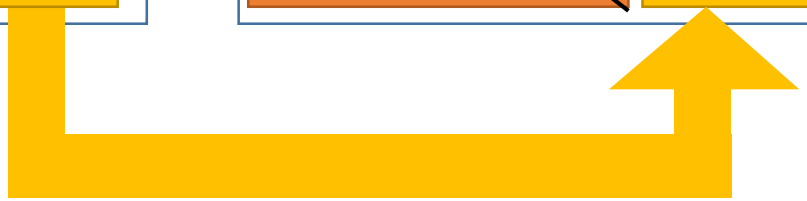
Retained Model-View

Stateless Model-View

Question 2:
How to do live programming?

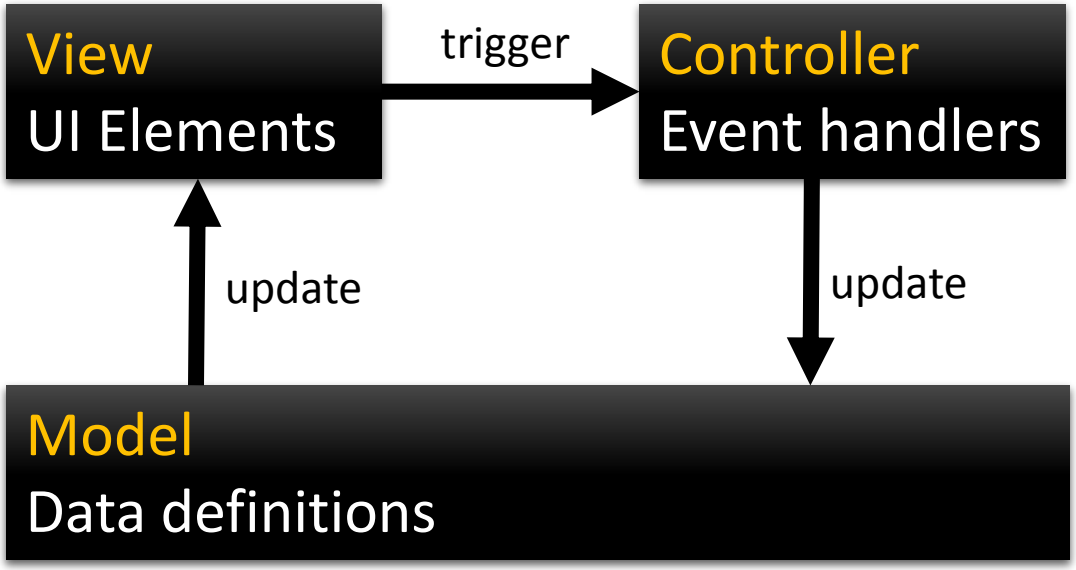
~~Replay Based~~

Model-View Based



Question 1: How to program GUIs?

- Model-View-Controller:
  Well established pattern for interactive applications
- Many variations exist



Question 1: How to program GUIs?

- Model-View-Controller:
  Well established pattern for interactive applications
- Many variations exist. We eliminate controller and put event handlers into the view.

View
UI Elements
Event Handlers

Model
Data definitions

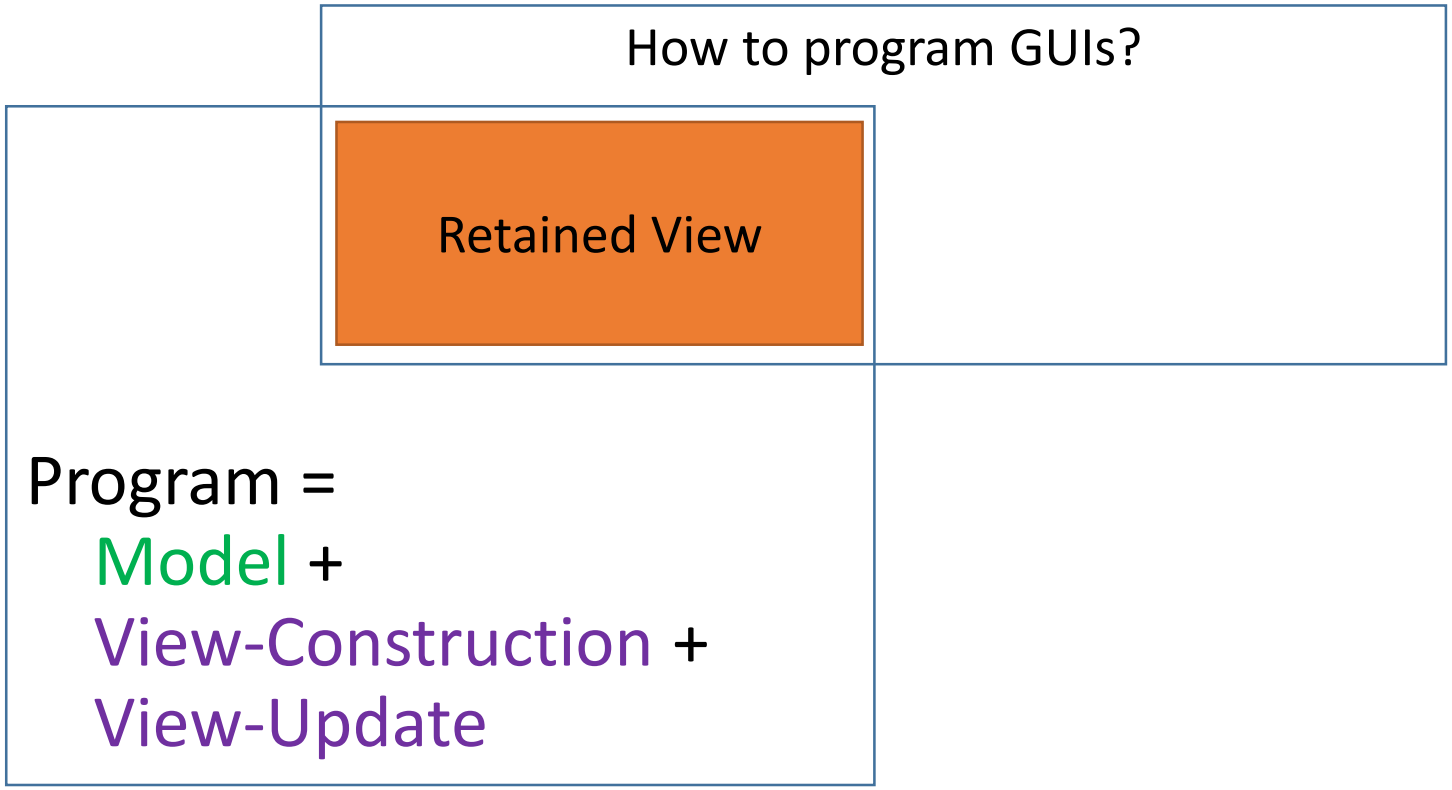
Question 1: How to program GUIs?

- Model-View-Controller:
  Well established pattern for interactive applications
- Many variations exist. We eliminate controller and put event handlers into the view.

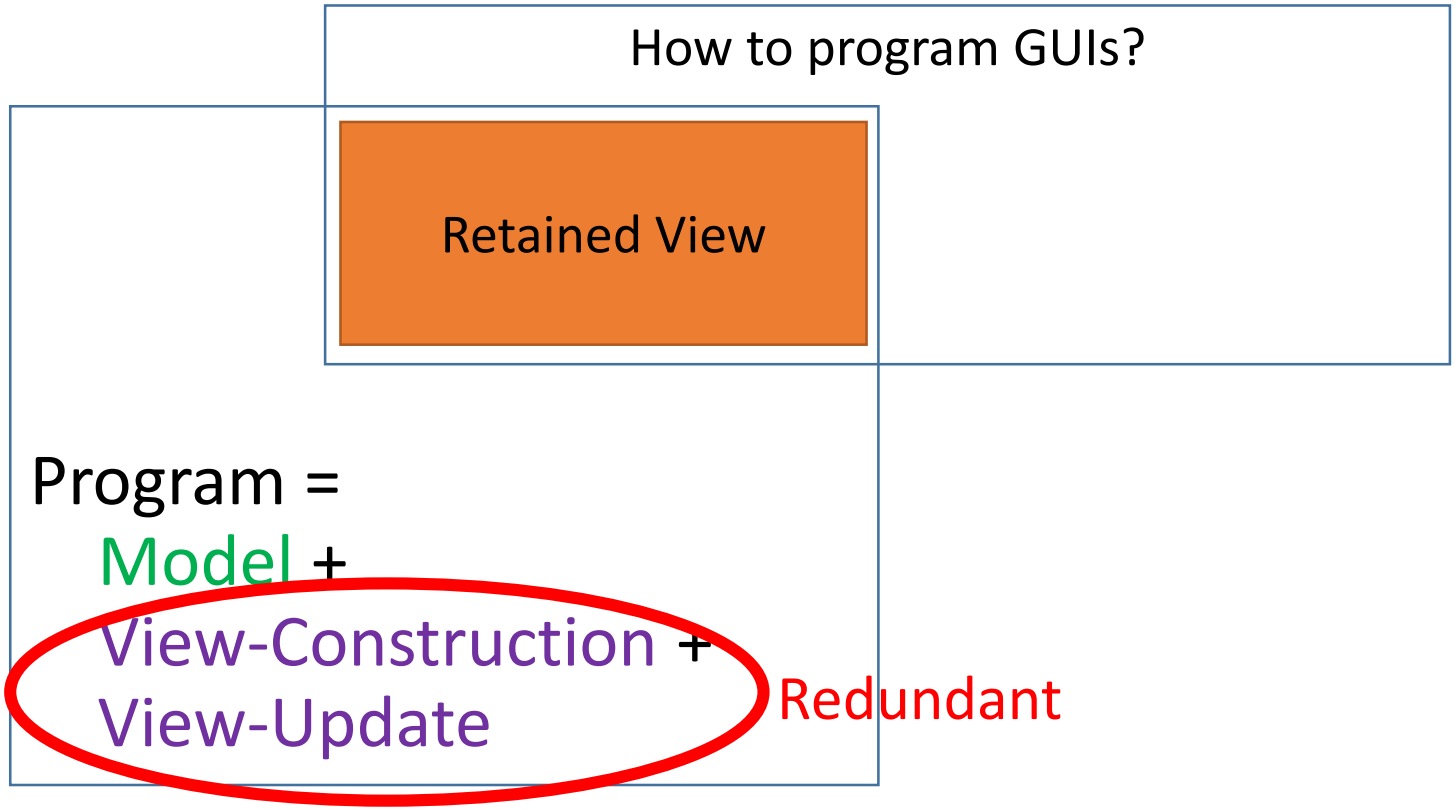
View
UI Elements
Event Handlers

Model
Data definitions

- Key question:
  How to define and maintain correspondence between view and model?

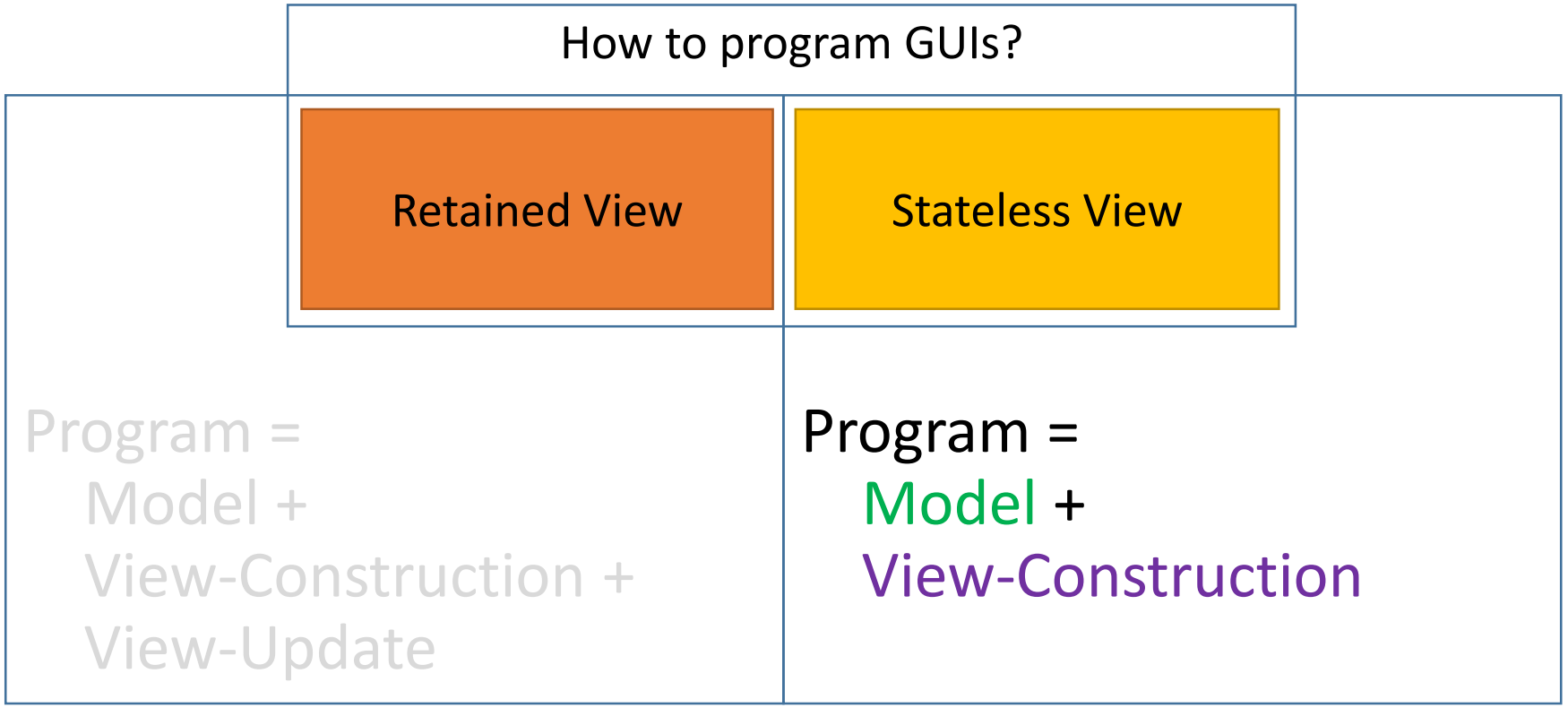


- Model: Data definitions that define the model
- View-Construction: Code that defines how to construct the view for a given model
- View-Update: Code that defines how to update the view in reaction to model changes



- Model: Data definitions that define the model
- View-Construction: Code that defines how to construct the view for a given model
- View-Update: Code that defines how to update the view in reaction to model changes

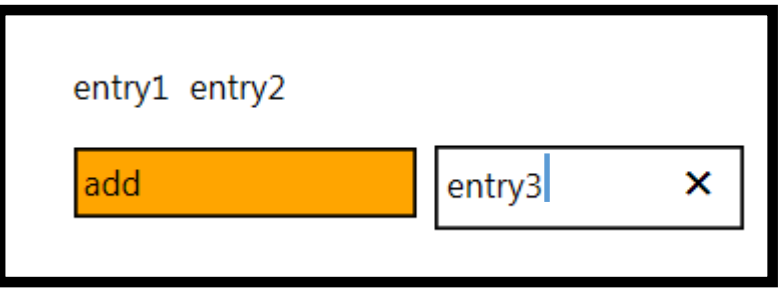
Error prone



- Model: Data definitions that define the model
- View-Construction: Code that defines how to construct the view for a given model

Update is simple: throw away old view, build new one.

Example.



Very simple app:
list of strings.

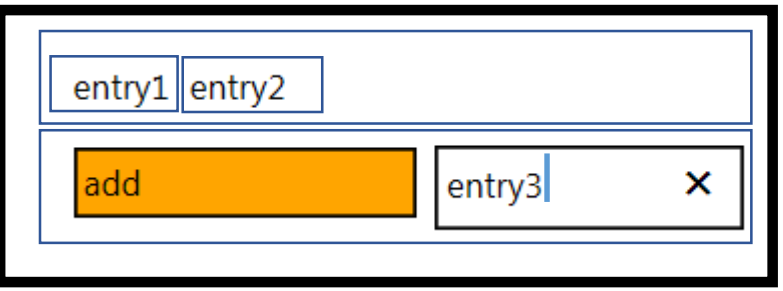
User can add entries by hitting the “add” button.

Program =

Model +

View-Construction

Example



Model

data entries: String Collection

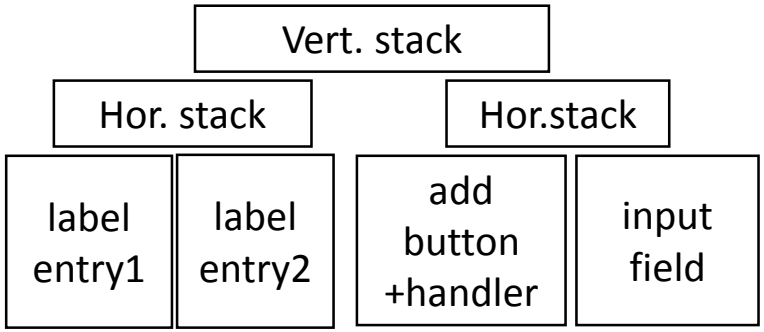
data field : String

View-Construction Example

Model

entries = [“entry1”, “entry2”]
field = “entry3”

Execute view construction code



View = Tree, decorated with attributes and event handlers

How to write view construction code?

Many frameworks are hybrids between a general-purpose language and a declarative language (e.g. C# + XML).

We would prefer: stay within single host language, but make code *look* as declarative as possible.

Host language for our prototype: TouchDevelop

Host language in the paper: lambda-calculus

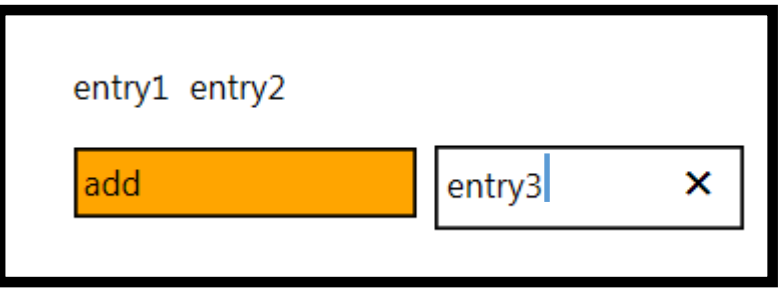
Idea: extend host language

- Special construct: nested “boxed” statements

```
boxed {
    .... nested code here....
}
```

- When executing, creates box tree implicitly
- view structure is implied by program structure, no need for programmer to manipulate collections!
- Code looks similar to declarative code.

Code Example.



View-Construction Code

```
display
    boxed
        box → use horizontal layout
        for each s in entries do
            boxed
                labelstyle()
                s → post
    boxed
        box → use horizontal layout
        boxed
            buttonstyle()
            "add" → post to wall
            on tapped(() => entries → add(field))
        boxed
            inputstyle()
            box → edit( field, (x) => field := x)

function buttonstyle()
    box → set border(colors → foreground, 0.1)
    box → set margins(0.5, 0.5, 0.5, 0.5)
    box → set padding(0.2, 0.2, 0.2, 0.2)
    box → set background(colors → orange)
    box → set width(10)
```

Model

data entries: String Collection

data field : String

No need for separate language or special collection classes.

- Adapt layout to various conditions – use a standard conditional
- Repeated elements - use standard loops
- Keep your code organized – use functional abstraction
- Provide widget collection – write a library

User interface element = just a function.

Question 1:
How to do live programming?

- This is now much easier to get a grasp on.

Question 1:

How to do live programming?

Answer:

on code changes, migrate model, build fresh view

on code changes, migrate model, build fresh view

Does Model Migration Work?

- Currently, we do something very simple
  - Variables whose types have changed are removed from model
- Experience: behaves reasonably in practice w.r.t to typical changes and user expectations
- More interesting solutions conceivable for structured data
  (cf. schema evolution, dynamic code updating)

on code changes, migrate model, build fresh view

Valid Concern: Speed?

- Isn't it too slow to reconstruct the view from scratch every time?

- In our experience (Browser-based, Javascript):
  - Re-executing the compiled display code is no problem for our apps (never more than 1000 objects on screen)
  - However, recreating the DOM tree from scratch is too slow (browser takes too much time) and has other issues (e.g. lose focus while typing in a textbox when it is replaced)
  - Fix: We implemented optimization that modifies the DOM tree incrementally when reexecuting the display code.

Yes, but what does all this mean, exactly?

- Paper contains a careful formalization of these concepts!
- Lambda calculus + UI primitives (boxes)
- Operational semantics
- System model for event-handling with page stack, UI, and code change events
- Type and Effect System

Expressions:

e	$::=$	v	(value)
	$\mid$	$e_1 e_2$	(application)
	$\mid$	f	(function)
	$\mid$	$(e_1, ..., e_n)$	(tuple), $(n \geq 0)$
	$\mid$	$e.n$	(projection), $(n \geq 1)$
	$\mid$	g	(read global)
	$\mid$	$g := e$	(write global)
	$\mid$	push p e	(push new page)
	$\mid$	pop	(pop page)
	$\mid$	boxed e	(create box)
	$\mid$	post e	(post content)
	$\mid$	box.$a := e$	(set box attribute)

Expressions:

Pure	e	$::= v$	(value)
		$e_1 e_2$	(application)
		f	(function)
		$(e_1, ..., e_n)$	(tuple), $(n \geq 0)$
		$e.n$	(projection), $(n \geq 1)$
Read/Write Model		g	(read global)
		$g := e$	(write global)
Navigation		push p e	(push new page)
		pop	(pop page)
View Construction		boxed e	(create box)
		post e	(post content)
		box.$a := e$	(set box attribute)

System Model

System State:

$$\sigma \quad ::= \quad (C, D, S, P, Q)$$

System Components:

C	$::=$	$\epsilon \mid C\ d$	(program code)
D	$::=$	$\perp \mid B$	(display)
S	$::=$	$\epsilon \mid S[g \mapsto v]$	(store)
P	$::=$	$\epsilon \mid P\ (p, v)$	(page stack)
Q	$::=$	$\epsilon \mid Q\ q$	(event queue)

Program Definitions:

d	$::=$	global $g : \tau = v$	(global)
	$\mid$	fun $f : \tau$ is e	(function)
	$\mid$	page $p(\tau)$ init e_1 render e_2	(page)

Box Content:

B	$::=$	ϵ	(empty)
	$\mid$	$B\ v$	(leaf content)
	$\mid$	$B\ [a = v]$	(box attribute)
	$\mid$	$B\ \langle B \rangle$	(nested box)

Events:

q	$::=$	[exec v]	(execute thunk)
	$\mid$	[push $p\ v$]	(push new page)
	$\mid$	[pop]	(pop page)

System Execution Steps

Three rules that enqueue events:

$$(\textsc{Startup})\ \frac{}{(C, D, S, \epsilon, \epsilon) \rightarrow_{\mathsf{g}} (C, \bot, S, \epsilon, [\mathsf{push\ start}\ ()])}$$

$$(\textsc{Tap})\ \frac{[\mathsf{ontap} = v] \in B}{(C, B, S, P, Q) \rightarrow_{\mathsf{g}} (C, \bot, S, P, [\mathsf{exec}\ v]\ Q)}$$

$$(\textsc{Back})\ \frac{}{(C, D, S, P, Q) \rightarrow_{\mathsf{g}} (C, \bot, S, P, [\mathsf{pop}]\ Q)}$$

Three rules that handle events:

$$(\textsc{Thunk})\ \frac{(C, S, Q, v\ ()) \rightarrow_{\mathsf{s}}^{*} (C, S', Q', ())}{(C, D, S, P, Q\ [\mathsf{exec}\ v]) \rightarrow_{\mathsf{g}} (C, \bot, S', P, Q')}$$

$$(\textsc{Push})\ \frac{C(p) = (f_i, f_r) \quad (C, S, Q, (f_i\ v)) \rightarrow_{\mathsf{s}}^{*} (C, S', Q', ())}{(C, D, S, P, Q\ [\mathsf{push}\ p\ v]) \rightarrow_{\mathsf{g}} (C, \bot, S', P\ (p, v), Q')}$$

$$(\textsc{Pop})\ \frac{P = P'(p, v) \quad \text{or} \quad P = P' = \epsilon}{(C, D, S, P, Q\ [\mathsf{pop}]) \rightarrow_{\mathsf{g}} (C, \bot, S, P', Q)}$$

One rule to refresh the display:

$$(\textsc{Render})\ \frac{C(p) = (f_i, f_r) \quad (C, S, \epsilon, (f_r\ v)) \rightarrow_{\mathsf{r}}^{*} (C, S, B, ())}{(C, \bot, S, P\ (p, v), \epsilon) \rightarrow_{\mathsf{g}} (C, B, S, P\ (p, v), \epsilon)}$$

One rule to change the program code:

$$(\textsc{Update})\ \frac{C' \vdash C' \quad C' : S \triangleright S' \quad C' : P \triangleright P'}{(C, D, S, P, \epsilon) \rightarrow_{\mathsf{g}} (C', \bot, S', P', \epsilon)}$$

Two execution modes with different allowed side effects

Event handler execution

- Can mutate model
- Can push/pop pages

$$\text{(ES-PURE)}\ \frac{(C,S,e) \to_{\mathsf{p}} (C,S,e')}{(C,S,Q,e) \to_{\mathsf{s}} (C,S,Q,e')}$$

$$\text{(ES-ASSIGN)}\ \frac{}{(C,S,Q,E[g := v]) \to_{\mathsf{s}} (C,S[g \mapsto v],Q,E[()])}$$

$$\text{(ES-PUSH)}\ \frac{}{(C,S,Q,E[\mathsf{push}\ p\ v]) \to_{\mathsf{s}} (C,S,[\mathsf{push}\ p\ v]\ Q,E[()])}$$

$$\text{(ES-POP)}\ \frac{}{(C,S,Q,E[\mathsf{pop}]) \to_{\mathsf{s}} (C,S,[\mathsf{pop}]\ Q,E[()])}$$

Display code execution

- Can set box attributes
- Can create boxes

$$\text{(ER-PURE)}\ \frac{(C,S,e) \to_{\mathsf{p}} (C,S,e')}{(C,S,B,e) \to_{\mathsf{r}} (C,S,B,e')}$$

$$\text{(ER-POST)}\ \frac{}{((C,S,B,E[\mathsf{post}\ v]) \to_{\mathsf{r}} (C,S,B\ v,E[()])}$$

$$\text{(ER-ATTR)}\ \frac{}{((C,S,B,E[\mathsf{box}.a := v]) \to_{\mathsf{r}} (C,S,B\ [a = v],E[()])}$$

$$\text{(ER-BOXED)}\ \frac{(C,S,\epsilon,e) \to_{\mathsf{r}}^{*} (C,S,B',v)}{(C,S,B,E[\mathsf{boxed}\ e]) \to_{\mathsf{r}} (C,S,B\ \langle B' \rangle,E[v])}$$

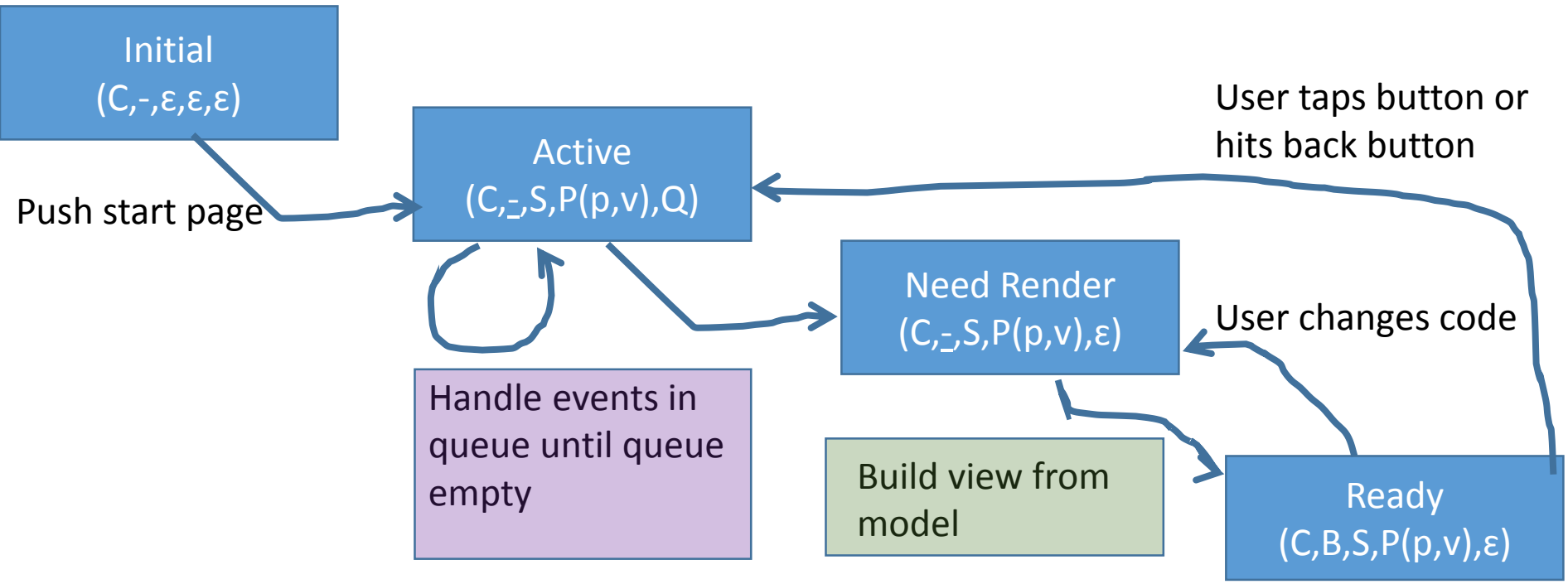
System Model Visualization

System State:

$$\sigma \quad ::= \quad (C, D, S, P, Q)$$

System Components:

C	$::=$	$\epsilon \mid C\ d$	(program code)
D	$::=$	$\perp \mid B$	(display)
S	$::=$	$\epsilon \mid S[g \mapsto v]$	(store)
P	$::=$	$\epsilon \mid P\ (p, v)$	(page stack)
Q	$::=$	$\epsilon \mid Q\ q$	(event queue)



Type & Effect System

- Judgments $\Gamma \vdash_{\mu} e : \tau$

μ ::= p | r | s pure, render, state effect

- Allows us to tell what kind of function we are looking at
- Lets us ensure that {event handlers, display code} only have the allowed side-effects for the given mode

Practical Experience

- Type/Effect system is sometimes too restrictive. For example, does not allow this in display code:

```
var x = new object();   x.field := value;
```

- More useful in practice: runtime checks that allow allocating fresh objects in a display heap, and allow mutation of the display heap

Goals

- Programming Model
  - Support succinct programming of apps with GUIs (graphical user interfaces)
  - Support live editing
  - Precise reactive semantics (user events, code changes)
- Implementation
  - Embed into TouchDevelop (language, runtime, IDE)
  - Enforce correct use of feature (separation of model and view)

Contributions

Live-View Approach

Formal System Model

Static Type/Effect System

Language Integration

Feature is public
Runs on all devices
touchdevelop.com