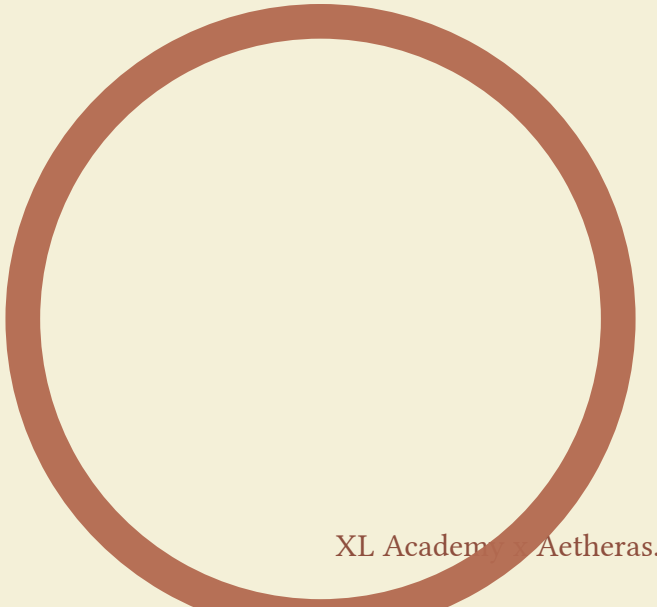
L16

AI Game Development & Computer Science Summer Program

HCI, Browser Interfaces, and UI Feedback



Two views of the same problem

A browser form and a game screen both answer: what can the user do now?

Browser interface

- Inputs collect intent.
- Validation prevents or explains mistakes.
- Feedback shows progress and results.

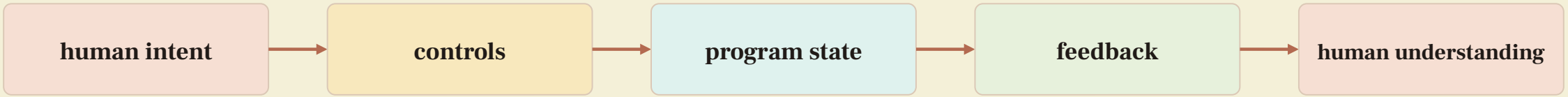
Game interface

- Controls collect intent.
- Rules update the game state.
- Feedback keeps the player oriented.

The core skill is the same: name the state, then make the screen match it.

An interface is a boundary

The user expresses intent; the computer communicates state and results.



Design questions

- What does the user need to know next?
- What can they safely do now?
- What mistake should the interface prevent or explain?

HCI is broader than screen layout

A good interface fits the human, the task, the context, and the machine.

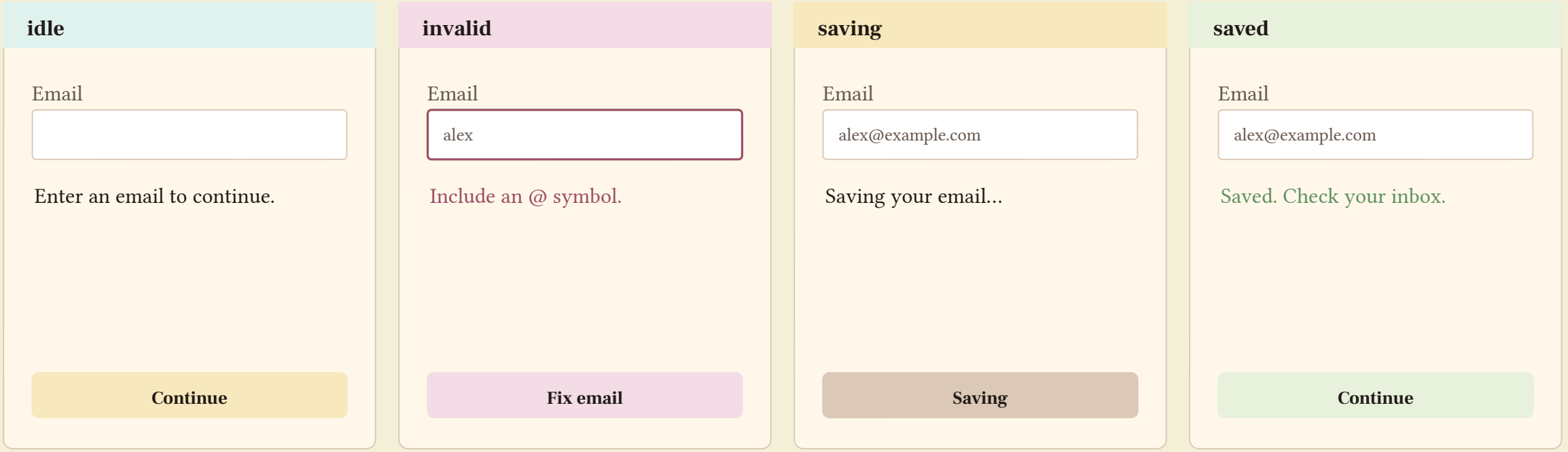
Part	Question to ask
human	What does the person know, need, fear, or expect?
task	What job are they trying to finish?
context	Where are they, how much time do they have, and what can interrupt?
computer	What can the system sense, compute, remember, and explain?
impact	What happens if the interface is wrong or too slow?

Design habit

- Do not start by asking where the button goes.
- Start by asking what interaction should become safer, clearer, or faster.

A form state is visible before it is code

The same feature changes message, controls, and next action.



Implementation habit: name the visible states before styling them.

A typed UI state keeps the interface honest

The type documents which states can exist.

```
type FormState =
  | { status: "idle" }
  | { status: "invalid"; message: string }
  | { status: "loading" }
  | { status: "success"; email: string };
```

Why this helps

- The success state must have an email.
- The invalid state must explain the problem.
- The loading state cannot also be success.

Exercise: audit a form state model

State type	Question to answer
{ status: "invalid" }	What message can we show?
{ status: "success" }	What did we save or submit?
{ loading: true, saved: true }	Can both be true at once?
{ status: "loading" }	What action should be disabled?

Rewrite one

- Pick the weakest row.
- Rewrite it as a safer union case.
- Say what the screen can render from it.

Feedback closes the loop

Every action should create a visible or understandable response.

Action	Weak feedback	Better feedback
submit invalid form	nothing changes	field message explains the fix
save request starts	button still looks normal	button says saving and is disabled
save succeeds	form disappears	confirmation states what happened

Affordance tells you what can be done

A control should look, read, and behave like its action.

Strong control

- Label starts with the action.
- Disabled state explains what is missing.
- Dangerous actions require enough friction.

Common bug

- A clickable thing does not look clickable.
- A disabled button gives no reason.
- A destructive action is too easy to trigger.

Friction is not always bad; it can prevent costly mistakes.

Accessibility is part of correctness

If the interface works only for one input style, the design is incomplete.

Need	Check
keyboard	reach and activate controls without a mouse
focus	current location is visible
labels	inputs have text labels
contrast	important text is readable
errors	message names the problem and the fix

Build habit

- Test with keyboard only before you call a form done.

Wait time changes behavior

Speed matters because people interpret delays as information.

Delay	Likely feeling	Interface response
about 0.1 sec	direct manipulation	show result
about 1 sec	still in flow	show progress if repeated
about 10 sec	attention drifts	set expectation or offer cancel

Design tradeoff

- A faster algorithm can improve usability.
- A slower task can still feel clear with honest progress feedback.

Observe users before trusting your intuition

Good HCI work alternates between ideas, prototypes, tests, and revision.

Observe

- Watch the task in context.
- Notice hesitation and workarounds.
- Ask what the user is trying to do.

Prototype

- Use a small, low-risk version.
- Make the main states visible.
- Test the flow before polishing.

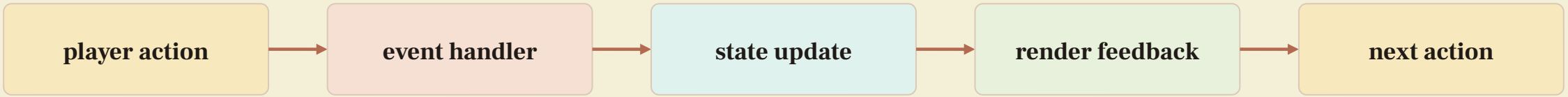
Revise

- Keep what helped.
- Cut what confused people.
- Update the state model and screen together.

Design loop: diverge with options, converge with evidence.

The same loop works for games

A game screen is still an interface between intent, rules, and feedback.



Trace habit

- Say the event.
- Say the old state.
- Say the new state.
- Say what the player sees next.

Trace one click all the way through

A concrete trace shows whether the feedback loop is actually responsive.

Moment	State	Visible feedback	Controls
before click	playing, score 4	Pick a card.	Play enabled
click handled	playing, score 6	Correct. +2 points.	Play enabled
target reached	won, score 10	You won.	Reset enabled
after reset	ready	Start a new round.	Start enabled

Talk through

- What event happened?
- Which state changed?
- What did the player see?
- Which action is safe next?

Exercise: trace a file check without running it

```
type FileCheck =
  | { status: "ready"; name: string }
  | { status: "blocked"; name: string; reason: string };

function reviewFile(name: string, sizeMB: number): FileCheck {
  if (sizeMB > 10)
    return { status: "blocked", name, reason: "too large" };
  return { status: "ready", name };
}
```

Trace cases

- reviewFile("avatar.png", 2)
- reviewFile("video.mov", 48)
- ready -> upload enabled
- blocked -> choose another file

Name the states the player can see

A state model is a small vocabulary for the game.

State	Visible feedback	Allowed action
ready	start screen	start
playing	score and prompt	play turn
won	success message	reset
lost	mistake message	try again

Rule

- If the player can see it, name it in the state model.

Use TypeScript to block impossible UI states

One union is clearer than many unrelated booleans.

```
type PlayState =
  | { status: "ready" }
  | { status: "playing"; score: number }
  | { status: "won"; score: number }
  | { status: "lost"; reason: string };
```

Trace it

- Pick one state.
- List the visible message.
- List the enabled controls.
- Decide the next valid event.

Feedback is more than text

The whole screen should agree about the current state.

ready

avatar.png is ready to upload.

2 MB file checked. Upload is available.

Upload | Choose file

blocked

video.mov is too large.

48 MB file. Choose a file under 10 MB.

Choose file | Cancel

Message, details, color, and enabled controls should all describe the same state.

Controls should match the state

State	Start	Play	Reset
ready	enabled	disabled	disabled
playing	disabled	enabled	enabled
won	disabled	disabled	enabled
lost	disabled	disabled	enabled

User-facing copy

- Say why a control is unavailable when it could confuse the player.

Mistakes need recovery paths

A mistake should teach the next step instead of ending the experience abruptly.

Problem	Better feedback
wrong move	explain the rule and allow retry
invalid setting	mark the field and keep previous value
game over	show score and reset action
confusing click	show which action is expected

Guard

- A player should always have a clear next step.

Exercise: design the recovery message

The message should say what happened and what can happen next.

State	Weak message	Your stronger message
{ status: "lost", reason: "tooManyMistakes" }	Game over.	
{ status: "invalid"; message: "name required" }	Error.	
{ status: "won"; score: 10 }	Done.	

A strong recovery message is specific, short, and tied to a valid next action.

Responsive design reduces repeated work

Do the smallest update that keeps the interface honest.

Instead of	Try
rerendering every element	update changed text
checking every rule in the handler	call a named helper
adding three messages	choose the clearest one

Design question

- What is the smallest visible change that answers the player's action?

AI can critique a flow, but it cannot observe your user

Use AI to widen your checklist, then test the actual interaction.

Useful prompt

- Here are my states, messages, and enabled controls.
- Find confusing labels.
- Suggest missing loading, error, and success states.
- List keyboard and accessibility checks.

Your review

- Reject generic advice.
- Check suggestions against your state model.
- Implement one small change and test it.

Build: a resilient sign-up form

```
type SignUpState =
  | { status: "idle" }
  | { status: "invalid"; message: string }
  | { status: "saving" }
  | { status: "saved"; email: string };
```

Build order

- Write the state type.
- Render each state.
- Add validation feedback.
- Check keyboard focus.
- Ask AI for one usability concern, then verify it.

Build: a game feedback panel

```
const render = (state: PlayState) => {
  score.textContent = "score" in state ? String(state.score) : "0";
  status.textContent = messageFor(state);
  playButton.disabled = state.status !== "playing";
  resetButton.disabled = state.status === "ready";
};
```

Build order

- Draw the visible states.
- Write the state type.
- Write one message helper.
- Wire one button.
- Ask AI for clearer feedback copy, then choose.

Exercise: write the message helper

```
function messageFor(state: PlayState): string {
  // ready: invite the player to start
  // playing: show the current score
  // won: celebrate and show the score
  // lost: explain the reason and next action
}
```

Check your helper

- Does every union case return a message?
- Does each message match the enabled controls?
- Does the lost state include a recovery path?

Proof checkpoint

Keep the artifact that proves today's concept.

Proof to keep

- One visible UI state model.
- One trace from event to state update to render.
- One disabled-control rule that matches the state.
- One recovery message for a mistake or failure.
- One AI critique you accepted, changed, or rejected with a reason.