OOP in SML

Yawar Raza

March 22, 2019

RIT Computer Science Community

Acknowledgements

- The Computer Science Community, for hosting this talk
- LaTeX using Beamer, for the slides
    - Metropolis theme with Fira fonts
    - `minted`, for code snippets
    - `pgf-umlcd`, for UML diagrams
- *Programming in Standard ML* by Robert Harper, where I learned how SML's module system works
- *Design Patterns* by the "Gang of Four", where I learned about object-oriented design
- The Dominion deck-building game, for our examples

Introduction

Overview

Standard ML is a programming language with support for imperative, functional, and modular programming, but it does *not* include OOP language constructs (like classes and interfaces).

We will learn how we can take object-oriented designs from languages like Java, and translate them into SML code that supports the same software design features.

This is *not* a general SML tutorial. We don’t show all the fundamental syntax, and we don’t cover any functional programming. Typical SML is written differently to how we’re writing it.

What I want you to get out of this talk

- Knowledge of some of SML's language constructs
    - SML has constructs that are much different than what you're used to. Learning about them will broaden your idea of how programs are written.
- A new perspective into what classes do
    - Classes provide more functionality than you might realize. Translating them to SML highlights all the different features classes provide.
- A bit of inspiration for your object-oriented designs
    - Some of the SML code may end up organized more clearly than the original Java code. You might want to organize your Java code the same way.

Imperative Programming

Implementing a simplified Dominion action card

Java

```java
class ActionCard {
    final int cards;
    final int actions;
    final int buys;
    final int coins;
}
```

SML

```sml
type actionCard = {
    cards: int,
    actions: int,
    buys: int,
    coins: int,
}
```

SML fields are *always* final, and must be initialized all at once:

```sml
val festival =
    { cards = 0, actions = 2, buys = 1, coins = 2 }
```

Estimate how good the card is

Java

```
int utility(ActionCard c) {
    return c.cards + c.actions +
        c.buys + c.coins;
}
```

SML

```
fun utility (c: actionCard) : int =
    (#cards c) + (#actions c) +
        (#buys c) + (#coins c)
```

`: actionCard` and `: int` are type annotations. They're often not needed in SML, but I'll write them most of the time here.

SML also supports “destructring” records, but we'll stick with field access notation as above.

Mutable local variables

In Dominion, players can play one action card, which sometimes gives them more actions to play more cards. (We're ignoring the other three card effects for this example.)

```
int actionsRemaining = 1;
while (actionsRemaining > 0) {
    Card card = /* player chooses a card */;
    actionsRemaining += card.actions;
}
```

Just like fields, local variables in SML are not mutable. So how do we write this in SML?

Reference cells

Reference cells are first-class mutable references. They are values just like ints and records are, so they can be assigned to local variables or stored as fields.

They have these three operations.

- `ref` *v* creates a new reference cell.
- `!r` retrieves the current value in the reference cell `r`.
- `r := ` *v* assigns a new value to the reference cell `r`.

`ref 1` has type `int ref`, which is like `ref<int>` with Java generics syntax.

Using reference cells

```
val actionsRemaining = ref 1
val () =
    while !actionsRemaining > 0 do
        let val card = (* player chooses a card *)
        in  actionsRemaining :=
                !actionsRemaining + (#actions card)
        end
```

Note that `actionsRemaining` always refers to the same ref cell; the ref cell’s internal value is what changes.

Ref cells are first-class values

Suppose we want to apply the card in a separate function. Here's how to do it in SML:

```
fun applyAction (c: actionCard, ar: int ref) =
    ar := !ar + (#actions c)

val actionsRemaining = ref 1
val () =
    while !actionsRemaining > 0 do
        let val card = (* player chooses a card *)
        in  applyAction (card, actionsRemaining)
        end
```

We can pass reference cells to functions and see their effects from the outside. We can't do this with a local variable in Java, but we can do this with a mutable object, via a wrapper class for `int`s.

Implementing reference cells in Java

This wrapper class, made generic, is exactly how SML's ref cells work.

```
class Ref<T> {
    private T obj;
    Ref(T initObj) { obj = initObj; }
    T get() { return obj; }
    void set(T newObj) { obj = newObj; }
}
```

Earlier we wrote a class `Card` for data aggregation. Here, we wrote a class `Ref` for mutability. We used the class construct for two *completely different* purposes!

In SML, we use records for data aggregation and ref cells for mutability. Using different constructs for different things makes both our code's behavior, and our intent, clearer.

Mutable fields

In Dominion, a player has a hand, a deck, and a discard pile, all of which change during the course of the game.

(Pretend we have a fully-featured `card` type, rather than the simplified one we wrote earlier.)

```
(* SML lists are immutable *)
type cardList = card list

type player = {
    hand: cardList ref,
    deck: cardList ref,
    discard: cardList ref
}
```

Now, let's implement a function for this record!

Mutating functions

When our deck becomes empty, we shuffle the discard pile, and turn that into our deck.

```
(* `unit` is like `void` in Java *)
fun replenishDeck (p: player) : unit = (
    (* no built-in shuffle; pretend we wrote one *)
    (#deck player) := shuffled !(#discard player);
    (#discard player) := []
)
```

Note that we mutated `p` without passing it as a ref cell. This “interior mutability” lets us easily write higher-level mutating functions on top of the primitive `:=`, just like in other imperative languages.

Constructors

Note that SML records do not have methods or constructors. `replenishDeck` is an example of how we would write a method. Below, `initialPlayer` is an example of a constructor:

```
val initialCards: cardList = (* ... *)
fun initialPlayer () : player =
    { hand = ref [], discard = ref [],
        deck = ref (shuffled initialCards) }
```

Constructors

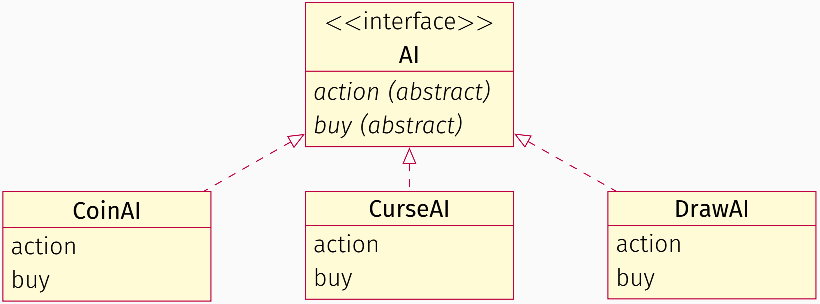
Players draw 5 cards at the start of the game. We can construct a player who's already drawn those cards:

```
fun drawN (p: player, n: int) = (* ... *)
fun startedPlayer () : player =
    let val p = initialPlayer ()
    in  (drawN (p, 5); p)
    end
```

Now, let's move on to some *actual* object-oriented programming.

Class Hierarchies

Strategies for computer-controlled Dominion opponents!



action performs the action phase of the AI’s turn and *buy* performs the buy phase of the AI’s turn.

CoinAI, **CurseAI**, and **DrawAI** are three different strategies the AI can take, each focusing on different elements of the game.

Java class skeletons

```
interface AI {
    void action(GameState gs);
    void buy(GameState gs);
}
class CoinAI implements AI {
    void action(GameState gs) { /* ... */ }
    void buy(GameState gs) { /* ... */ }
}
class CurseAI implements AI {
    void action(GameState gs) { /* ... */ }
    void buy(GameState gs) { /* ... */ }
}
class DrawAI implements AI {
    void action(GameState gs) { /* ... */ }
    void buy(GameState gs) { /* ... */ }
}
```

A type corresponds to a set of objects

$$\begin{aligned}\text{set of } \textsf{AI} \text{ objects} = {} & (\text{set of } \textsf{CoinAI} \text{ objects}) \\ \cup {} & (\text{set of } \textsf{CurseAI} \text{ objects}) \\ \cup {} & (\text{set of } \textsf{DrawAI} \text{ objects})\end{aligned}$$

Furthermore, CoinAI, CurseAI, and DrawAI are disjoint from each other, meaning that the unions above are *disjoint* unions.

Tagged unions

```
(* record fields omitted *)
type CoinAI = { (* ... *) }
type CurseAI = { (* ... *) }
type DrawAI = { (* ... *) }

datatype AI =
    COIN_AI of CoinAI
  | CURSE_AI of CurseAI
  | DRAW_AI of DrawAI
```

No reflection in SML; cannot query types at runtime. Can only query *tags* like `COIN_AI`, `CURSE_AI`, and `DRAW_AI`.

This makes types solely a compile-time construct, while tags are solely a run-time construct. Different constructs for different uses.

Pattern matching

```
fun CoinAI_action (coinAI: CoinAI, gs: GameState) = (* ... *)
fun CurseAI_action (curseAI: CurseAI, gs: GameState) = (* ...
fun DrawAI_action (drawAI: DrawAI, gs: GameState) = (* ... *)

fun AI_action (ai: AI, gs: GameState) =
    case ai of
        COIN_AI (coinAI: CoinAI) =>
            CoinAI_action (coinAI, gs)

      | CURSE_AI (curseAI: CurseAI) =>
            CurseAI_action (curseAI, gs)

      | DRAW_AI (drawAI: DrawAI) =>
            DrawAI_action (drawAI, gs)
```

`_action` is not special syntax, but just a naming convention we're using here for clarity (not seen in real SML code).

Upcasting: implicit vs explicit

Java

```java
DrawAI drawAI = /* ... */;
AI someAI = drawAI;
```

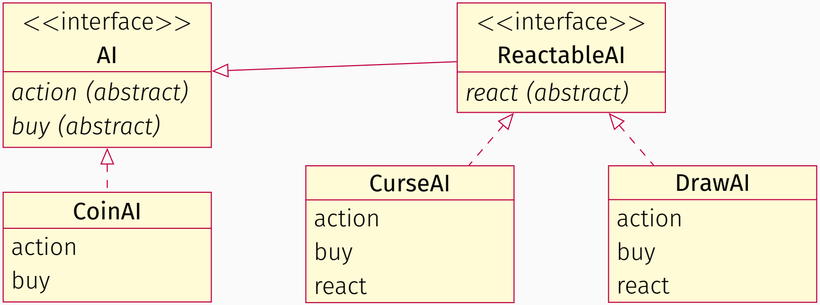
SML

```sml
val (drawAI: DrawAI) = (* ... *)
val (someAI: AI) = DRAW_AI drawAI
```

Converting to and from the interface type must be done explicitly through tag application and pattern matching respectively.

Subinterfaces

In Dominion, when players can attack other players, they can counter with Reaction cards. We'll add support for this to some of our AIs.



$$\text{ReactableAI} = \text{CurseAI} \cup \text{DrawAI}$$

$$\text{AI} = \text{CoinAI} \cup \text{ReactableAI}$$

Java class skeletons

```java
interface AI { /* ... */ }
class CoinAI implements AI { /* ... */ }

interface ReactableAI extends AI {
    void react(Card c, GameState gs);
}
class CurseAI implements ReactableAI { /* ... */ }
class DrawAI implements ReactableAI { /* ... */ }
```

Nested tagged unions

```
type CoinAI = { (* ... *) }
type CurseAI = { (* ... *) }
type DrawAI = { (* ... *) }

datatype ReactableAI =
    CURSE_AI of CurseAI | DRAW_AI of DrawAI

datatype AI =
    COIN_AI of CoinAI | REACTABLE_AI of ReactableAI
```

Pattern matching

```
fun CoinAI_action (coinAI: CoinAI, gs: GameState) = (* ... *)
fun CurseAI_action (curseAI: CurseAI, gs: GameState) = (* ...
fun DrawAI_action (drawAI: DrawAI, gs: GameState) = (* ... *)

fun ReactableAI_action (rAI: ReactableAI, gs: GameState) =
    case rAI of
        CURSE_AI (curseAI) =>
            CurseAI_action (curseAI, gs)
      | DRAW_AI (drawAI) =>
            DrawAI_action (drawAI, gs)

fun AI_action (ai: AI, gs: GameState) =
    case ai of
        COIN_AI (coinAI) =>
            CoinAI_action (coinAI, gs)
      | REACTABLE_AI (rAI) =>
            ReactableAI_action (rAI, gs)
```

Transitive upcasting

Java

```
DrawAI drawAI = /* ... */;
AI someAI = drawAI;
```

SML

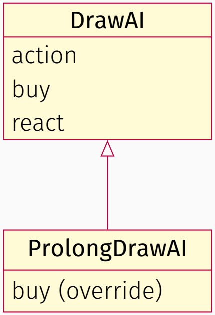
```
val (drawAI: DrawAI) = (* ... *)
val (someAI: AI) =
    REACTABLE_AI (DRAW_AI drawAI)
```

DrawAI and **AI** are only related through **ReactableAI**, so converting to and from **DrawAI** and **AI** must always go through **ReactableAI**.

We can write functions to hide this intermediate step, which directly convert between **DrawAI** and **AI**.

Implementation inheritance

In Dominion, buying cards can sometimes end the game, so an AI might avoid buying those cards if they have a long-term strategy.



```
class DrawAI { /* ... */ }
class ProlongDrawAI extends DrawAI {
    @Override
    void buy(GameState gs) {
        /* ... */
    }
}
```

The sets that these types correspond to

$$\begin{aligned} &\text{set of objects with } \textit{type}\ \textsf{DrawAI} = \\ &\quad(\text{set of objects } \textit{instantiated} \text{ from } \textit{class}\ \textsf{DrawAI}) \\ &\cup(\text{set of objects } \textit{instantiated} \text{ from } \textit{class}\ \textsf{ProlongDrawAI}) \end{aligned}$$

DrawAI refers to two different things, a type and an instantiating class. How confusing! What if we called the type AnyDrawAI and the instantiating class DefaultDrawAI?

$$\begin{aligned} &\text{set of objects with } \textit{type}\ \textsf{AnyDrawAI} = \\ &\quad(\text{set of objects } \textit{instantiated} \text{ from } \textit{class}\ \textsf{DefaultDrawAI}) \\ &\cup(\text{set of objects } \textit{instantiated} \text{ from } \textit{class}\ \textsf{ProlongDrawAI}) \end{aligned}$$

Can we do this in Java as well?

Subtyping without class inheritance

```
interface AnyDrawAI {
    void action(GameState gs);
    void buy(GameState gs);
    void react(Card c, GameState gs);
}
class DefaultDrawAI implements AnyDrawAI {
    /* code is from old DrawAI class */
}
class ProlongDrawAI implements AnyDrawAI {
    /* code is discussed on next slide */
}
```

Now **AnyDrawAI** corresponds to the same set of objects that **DrawAI** did before. Any object from the old hierarchy has a corresponding one in the new hierarchy.

Code reuse without class inheritance

ProlongDrawAI is supposed to inherit *action* and *react* from DefaultDrawAI. Use *delegation* to accomplish this instead.

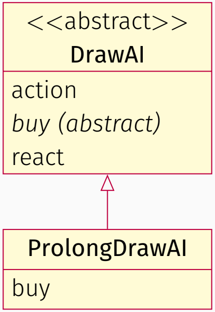
```java
class ProlongDrawAI implements AnyDrawAI {
    private DefaultDrawAI superObj;

    ProlongDrawAI(/* ... */) {
        // super constructor call
        superObj = new DefaultDrawAI(/* ... */);
    }
    void react(Card c, GameState gs) {
        // super method call
        superObj.react(c, gs);
    }
    // more fields and methods
}
```

Now we can translate it to SML

```
type DefaultDrawAI = { (* ... *) }
type ProlongDrawAI = {
    superObj: DefaultDrawAI,
    (* ... *)
}

datatype AnyDrawAI =
    DEFAULT_DRAW_AI of DefaultDrawAI
  | PROLONG_DRAW_AI of ProlongDrawAI

fun ProlongDrawAI_react (pdAI, c, gs) =
    DefaultDrawAI_react ((#superObj pdAI), c, gs)

(* the other method implementations *)
```

Abstract classes

What if we started out with this instead?



```
abstract class DrawAI {
    abstract void buy(GameState gs);
    /* ... */
}
class ProlongDrawAI extends DrawAI {
    void buy(GameState gs) {
        /* ... */
    }
}
```

What changes (and what doesn't)

```
type DefaultDrawAI = { (* ... *) }
type ProlongDrawAI = {
    superObj: DefaultDrawAI,
    (* ... *)
}

(* remove DEFAULT_DRAW_AI case *)
(* also remove it from AnyDrawAI functions *)
datatype AnyDrawAI =
 (* DEFAULT_DRAW_AI of DefaultDrawAI | *)
    PROLONG_DRAW_AI of ProlongDrawAI

fun ProlongDrawAI_react (pdAI, c, gs) =
    DefaultDrawAI_react ((#superObj pdAI), c, gs)

(* and remove DefaultDrawAI_buy (duh!) *)
```

Q&A time

- Why remove `DEFAULT_DRAW_AI`?
    - Since **DrawAI** is now an abstract class, there aren't instances of it. So **AnyDrawAI** shouldn't include instances of **DefaultDrawAI**.
- So why keep `type DefaultDrawAI = { (* ... *) }`?
    - It is still used for the *subobjects* included in **ProlongDrawAI**, like it was before.
- And what about `DefaultDrawAI_action` and `DefaultDrawAI_react`?
    - Rather than satisfying an interface, they simply provide a toolbox of implementations. They are provided for subclasses to freely use or ignore as desired.

Refactoring

DefaultActionAI isn't even used as an AI anymore! It just provides *action* and *react* strategies for other AIs.

We could even split up the two strategies so they can be used independently. In ProlongDrawAI, we'd replace `superObj` with them:

```
type ProlongDrawAI = {
    actionStrategy: AddActionsThenAddCards,
    reactStrategy: OnlyPreventDiscardAttacks,
    (* ... *)
}
```

`ProlongDrawAI_action` and `ProlongDrawAI_react` would change to use these fields. Using these functions remains the same.

But what if someone had been accessing `#superObj` directly? Can we make it **`private`**, like in Java? Actually, SML does this differently...

Encapsulation and Extensibility

Record fields are public

Remember this type?

```
type player = {
    hand: cardList ref,
    deck: cardList ref,
    discard: cardList ref
}
```

What if we want to change it later to use immutable arrays instead of linked lists, or to keep the hand organized by card category?

To support this, we need to hide the record fields, and only expose the higher-level behavior involving `player`s.

Player functions

First, we should determine what operations we want to expose. For simplicity, we won't support actually playing cards, just drawing and discarding them. The functions we then need are:

- `drawN`: Draw *n* cards from the deck
- `discardHand`: Discard the entire hand
- `startedPlayer`: Create a new player who's already drawn their starting hand

In Java, these would be the **`public`** methods of the `Player` class, while the fields and other methods would be **`private`**.

Signatures

SML uses *signatures* to specify which operations a *structure* (or module) should expose. Signatures and structures are defined separately from each other.

```
signature PLAYER_DATA =
sig
    type player

    val drawN : player * int -> unit
    val discardHand : player -> unit
    val startedPlayer : unit -> player
end
```

We expose the *name* of the type `player` (but not its implementation), along with functions that refer to this type.

The implementation goes in a structure

```
structure PlayerData :> PLAYER_DATA =
struct
    type cardList = card list (* not exported *)
    type player = { (* ... *) }

    (* hidden functions *)
    fun shuffled (l: cardList) : cardList = (* ... *)
    fun replenishDeck (p: player) : unit = (* ... *)
    fun initialPlayer () : player = (* ... *)

    (* exported functions *)
    fun drawN (p: player, n: int) : unit = (* ... *)
    fun discardHand (p: player) : unit = (* ... *)
    fun startedPlayer () : player = (* ... *)
end
```

Encapsulation in action

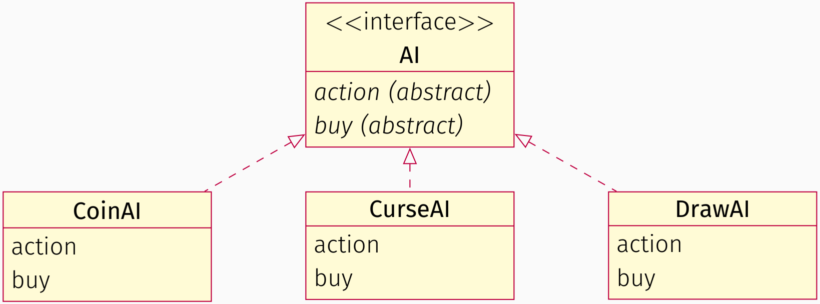
The commented out lines won't work because those implementation details aren't exposed by the signature.

```
(* give module a shorter alias *)
structure PD = PlayerData

(* val (p: player) = { (* ... *) } *)
(* val (p: player) = PD.initialPlayer () *)
val (p: player) = PD.startedPlayer ()

(* val () = (#hand p) := [] *)
val () = PD.discardHand (p)
```

We call `player` an *abstract* (or *opaque*) type. The only operations we can do on it are the ones exported by the module.

Encapsulating a class hierarchy

Recall our first example of a class hierarchy:



What if we wanted to replace the **DrawAI** class with the **AnyDrawAI** hierarchy we created earlier? We shouldn't expose the concrete classes, just the **AI** interface.

Desired signature

(Another benefit is that we can expose AIs by difficulty instead of by strategy, if we want.)

```
signature AI_SIG =
sig
    type AI

    val easyAI : unit -> AI
    val mediumAI : unit -> AI
    val hardAI : unit -> AI

    val action : AI -> GameState -> unit
    val buy : AI -> GameState -> unit
end
```

The implementing structure (part 1 — hidden internals)

```
structure AIModule :> AI_SIG =
struct
    type CoinAI = { (* ... *) }
    type CurseAI = { (* ... *) }
    type DrawAI = { (* ... *) }

    fun mkCoinAI () : CoinAI = (* ... *)
    fun mkCurseAI () : CurseAI = (* ... *)
    fun mkDrawAI () : DrawAI = (* ... *)

    fun CoinAI_action (coinAI: CoinAI, gs) = (* ... *)
    fun CurseAI_action (curseAI: CurseAI, gs) = (* ... *)
    fun DrawAI_action (drawAI: DrawAI, gs) = (* ... *)

    fun CoinAI_buy (coinAI: CoinAI, gs) = (* ... *)
    fun CurseAI_buy (curseAI: CurseAI, gs) = (* ... *)
    fun DrawAI_buy (drawAI: DrawAI, gs) = (* ... *)
```

The implementing structure (part 2 — exposed API)

```
    datatype AI = COIN_AI of CoinAI
        | CURSE_AI of CurseAI | DRAW_AI of DrawAI

    fun easyAI () : AI = COIN_AI (mkCoinAI ())
    fun mediumAI () : AI = CURSE_AI (mkCurseAI ())
    fun hardAI () : AI = DRAW_AI (mkDrawAI ())

    fun action (ai: AI, gs: GameState) =
        case ai of
            COIN_AI (coinAI: CoinAI) =>
                CoinAI_action (coinAI, gs)
          | CURSE_AI (curseAI) => (* ... *)
          | DRAW_AI (drawAI) => (* ... *)

    fun buy (ai: AI, gs: GameState) = (* ... *)
end
```

Using the AI interface in Java

What if we wanted to write code that runs a game between two AIs? In Java, we might use a class like this one:

```java
class TwoPlayerGame {
    AI ai1;
    AI ai2;
    GameState gs;

    TwoPlayerGame(AI ai1, AI ai2) { /* ... */ }

    void runRound() {
        ai1.action(gs);
        ai1.buy(gs);
        a12.action(gs);
        ai2.buy(gs);
    }
}
```

Using the AI type in SML

```
structure TpgMod = (* no sig means nothing hidden *)
struct
    structure M = AIModule

    type TwoPlayerGame =
        { ai1: M.AI, ai2: M.AI, gs: GameState }
    fun makeGame (ai1: M.AI, ai2: M.AI) = (* ... *)

    fun runRound (game: TwoPlayerGame) = (
        M.action (#ai1 game, #gs game);
        M.buy (#ai1 game, #gs game);
        M.action (#ai2 game, #gs game);
        M.buy (#ai2 game, #gs game)
    )
end
```

Creating a new AI

Suppose the previous AI hierarchy was provided as a library to us (such as a Java package), but we want to create a new AI strategy that only focuses on collecting victory cards. In Java, we can write a new class that implements **AI**.

```
class VictoryAI implements AI { /* ... */ }
```

And use it with our existing **TwoPlayerGame** class.

```
new TwoPlayerGame(new VictoryAI(), new VictoryAI())
```

But SML doesn't let us extend the `M.AI` type. Tagged unions are *closed*, and besides, we've hidden the fact that it even is a tagged union. How do we allow for new AIs in our SML code?

"Versions" of a class hierarchy

We can think of the AI hierarchy provided by the original library as a sort of *Version 1*.

AI hierarchy: Version 1

Implementing classes: CoinAI, CurseAI, DrawAI

When we, a client of the library, implement the AI interface with a new class, we create a *Version 2*.

AI hierarchy: Version 2

Implementing classes: CoinAI, CurseAI, DrawAI, VictoryAI

When we use AI in the TwoPlayerGame class, it works for *any version* of the hierarchy. The SML module `TpgMod`, however, works only for Version 1 because it uses `AIModule` (via alias `M`) directly.

Another signature

Our AI_SIG signature encapsulates the AI hierarchy as a whole, but TwoPlayerGame only depends on the AI interface. We write a new signature exposing just the interface methods.

```
signature AI_IFACE_SIG =
sig
    type AI

    val action : AI -> GameState -> unit
    val buy : AI -> GameState -> unit
end
```

type AI can refer to any "version" of the interface. Thus, the API we're specifying is decoupled from the set of objects that type AI corresponds to. In Java, these are both tied to interfaces.

Functors: parameterized structures

Now we can parameterize the structure to work on any version of the hierarchy the user wants. SML supports this using *functors*:

```
functor TpgFn (P : AI_IFACE_SIG) =
struct
    type TwoPlayerGame =
        { ai1: P.AI, ai2: P.AI, gs: GameState }
    fun createGame (ai1: P.AI, ai2: P.AI) = (* ... *)
    fun runRound (game: TwoPlayerGame) = (* ... *)
end

(* how to get the old TpgMod again *)
structure TpgV1 = TpgFn (AIModule)
```

Note that modules don't need to be declared with a signature in order to satisfy it, which is why `AIModule` satisfies `AI_IFACE_SIG`.

How to create “Version 2” of the AI hierarchy

By adding a new class VictoryAI that implements AI, we’re adding new objects to the set of objects of type AI. Thinking in terms of versions, taking “snapshots” of this set, this means:

$$\begin{aligned}\text{set of } \mathsf{AI}_{V2} \text{ objects} = {} & (\text{set of } \mathsf{AI}_{V1} \text{ objects}) \\ & \cup (\text{set of } \mathsf{VictoryAI} \text{ objects})\end{aligned}$$

This suggests using a tagged union for our new AI type.

In previous examples, the sets being unioned were either classes implementing the interface, or subinterfaces extending the interface. Here, one of the sets is instead a previous version of the interface.

Code for “Version 2” (part 1 — setup)

```
(* new signature to expose new constructor *)
signature AI_V2_SIG =
sig
    include AI_SIG

    fun naiveAI : unit -> AI
end

structure AIModuleV2 :> AI_V2_SIG =
struct
    structure Old = AIModule

    type VictoryAI = { (* ... *) }
    fun mkVictoryAI () : VictoryAI = (* ... *)
    fun VictoryAI_action (vAI: VictoryAI, gs) = (* ... *)
    fun VictoryAI_buy (vAI: VictoryAI, gs) = (* ... *)
```

Code for “Version 2” (part 2 — uniting old and new)

```
    datatype AI =
        OLD_AI of Old.AI | VICTORY_AI of VictoryAI

    fun easyAI () : AI = OLD_AI (Old.easyAI ())
    fun mediumAI () : AI = OLD_AI (Old.mediumAI ())
    fun hardAI () : AI = OLD_AI (Old.hardAI ())
    fun naiveAI () : AI = VICTORY_AI (mkVictoryAI ())

    fun action (ai: AI, gs: GameState) =
        case ai of
            OLD_AI (oldAI: Old.AI) =>
                Old.action (oldAI, gs)
          | VICTORY_AI (victoryAI: VictoryAI) =>
                VictoryAI_action (victoryAI, gs)

    fun buy (ai: AI, gs: GameState) = (* ... *)
end
```

Going back to our functor

If you're ready to commit to `AIModuleV2` (like in an end-user application):

```
structure TpgV2 = TpgFn (AIModuleV2)
```

But if you want to be open to more extensions (like in a library), wrap it inside another functor:

```
functor Client (P: AI_IFACE_SIG) =
struct
    structure TpgVn = TpgFn (P)
    (* can call TpgVn.runRound, etc. *)
end
```

By doing this, we can keep our programs as general as possible for as long as we need.

Conclusion

What I wanted you to get out of this talk

- Knowledge of some of SML's language constructs
    - We introduced reference cells, tagged unions, and the module system, giving new insight to the simple ideas of mutation, types, and encapsulation.
- A new perspective into what classes do
    - We identified the specific features that classes provide, and showed how each feature can be explicitly added in our translated SML code.
- A bit of inspiration for your object-oriented designs
    - We used composition to replace inheritance, which is a good idea in Java as well. Did you get any more inspiration from the SML code? Let me know!