Case Inference

In this article we will talk about case inference, syntax and semantic isomorphism, and how understanding these things can help you become a better programmer.

Don't worry these are all just fancy names for simple concepts.

Naming conventions

One of the most important tasks in programming is picking good names for your identifiers (constants, variables, functions, classes...)

This is often hard because some concepts cannot be described with one word so we use word concatenations: list_iterator, robotControl, DatabaseConnection, PlainDocumentModel … Because there are multiple ways of marking this concatenations, coding conventions were invented.

In the OO world it seems that a prevailing convention is to write class names in PascalCase, identifier names in camelCase and constants in UPPERCASE_WITH_UNDERSCORES

No matter what convention you use it is always a good practice to name related things in a manner that makes it obvious they are related. Programmers do this subconsciously all the time.

Naming object variables like class name:
```
var spaceShuttle = new SpaceShuttle();
```

Naming constants like their value:
```
constant String ID_PERSON  = "person";
```

Naming resources like file name:
```
Icon file_open = loadIcon("resources/file-open.png");
```

Isomorphism

Two thing are isomorphic if they can be transformed one to another.

Syntax Isomorphisms	Semantic Isomorphisms
`north_west ↔ northWest`	`north_west ↔ up_left`
`north_east ↔ northEast`	`north_east ↔ up_right`
`south_west ↔ southWest`	`south_west ↔ down_left`
`south_east ↔ southEast`	`south_east ↔ down_right`

Syntax transformation in example above is simple **_x ↔ X** where x is any character.
Syntax transformations can always be implemented with one rule.

Semantic transformation is not that simple, it requires mapping of names:
[north ↔ up, south ↔ down, west ↔ left, east ↔ right]
We got four mappings here, same as in the table above so we might as well not bother.
Semantic transformations often require verbatim copies = one rule for every list entry.

Identifier "CASE"

Identifier case can be:

- `PascalCaseIdentifier`
- `camelCaseLikeInJava`
- `c_style_identifier`
- `C_STYLE_CONSTANT`
- `any%%custom%%separator%%case`
- `noseparatornocharactercasealteringhardtoreadcase`

All identifier cases except the last one are syntax isomorphs, they can be translated to one another with simple rules. To convert between pascal and camel case you only need to change the character case of first character:

```
NorthWest ↔ northWest
```

To convert c_style_id into camelCase you need to delete all underscores and uppercase the character after them, to do the opposite you find all uppercase characters, lower case them and put an underscore in front:

```
north_west ↔ northWest
```

To convert between c_style_id and custom#@#separator#@#case you replace underscore with custom separator:

```
north_west ↔ north#@#west
```

And so on....
Notice that adding common prefixes or suffixes to names does not destroy syntax isomorphism.

```
ID_PERSON       ↔    person     ↔    personId
ID_PHONE_NUM    ↔    phone-num  ↔    phoneNumId
```

All this conversions can be implemented with regular expressions and therefore they can be automated, and if something can be automated than it should be, but we will get to that later.

The last case that I personally call "stupidcase" is not a syntax but a semantic isomorph.

Converting northwest to north_west might be easy for English speakers but it is impossible for computers because it requires semantic context (knowledge of words).

If you do not believe me try converting these to c_style:

- `FLUGZEUGABWEHRKANONE`
- `PROTIVVAZDUŠNITOP`
- `ILMAPUOLUSTUKSENGUN`

Btw this is all same term in three different languages.

Now let use see why is it important not to use stupidcase in your programs, read on.

Repetitive code

Isomorphic naming often leads to code like this:

```
case NORTH_WEST:
      robot.goNorthWest();
      break;
case NORTH:
      robot.goNorth();
      break;
case NORTH_EAST:
      robot.goNorthEest();
      break;
```

or to code like this:

```
Icon FILE_OPEN    = loadIcon("resources/file-open.png");
Icon FILE_CLOSE   = loadIcon("resources/file-close.png");
Icon FILE_SAVE    = loadIcon("resources/file-save.png");
Icon FILE_SAVE_AS = loadIcon("resources/file-save-as.png");
```

or to code like this:

```
Action CUT    = new Action("Cut",    ICON_CUT);
Action COPY   = new Action("Copy",   ICON_COPY);
Action PASTE  = new Action("Paste",  ICON_PASTE);
Action DELETE = new Action("Delete", ICON_DELETE);
```

What is really annoying about this type of code is all the extra typing you must do when you know it could be simply generated from a list of names. Simple past/replace does not work either because there are different cases of same identifier involved and you know it is an overkill to make code generator for this kind of one-time job so you keep typing...

and you sigh and think to yourself...

if only there was a tool that could guess the case of identifiers...

a kind of case inference...

and generate blocks of code from a simple example and a list of names.

`Action CUT = new Action("Cut", ICON_CUT);`	**cut**
	copy
	paste
	delete
	...

But enough with the commercials, we are not done with theory yet.

Unnecessary semantics

Some people write code like this:

```
      case NORTH_WEST:
            robot.goUpLeft();
            break;
      case NORTH:
            robot.goUp();
            break;
      case NORTH_EAST:
            robot.goUpRight();
            break;
```

I personally think they should be shot as an example for junior programmers. Introducing semantic translations when they are not needed is error prone and misleading. In the example above you should either rename your constants or your method names.

Necessary semantics

But sometimes code contains mappings that are semantic by nature.
Mapping of actions to hotkeys is an example of pure semantic linking:

```
Action CUT    = new Action("Cut",    Ctrl-X);
Action COPY   = new Action("Copy",   Ctrl-C);
Action PASTE  = new Action("Paste",  Ctrl-V);
Action DELETE = new Action("Delete", DEL);
```

Now we need two lists to generate this code. Unlike the first list that is smart and uses case inference to generate all occurrences of syntax isomorphism, second one just does verbatim copying.

`Action CUT = new Action("Cut", Ctrl-X);`	**cut**	**Ctrl-X**
	copy	Ctrl-C
	paste	Ctrl-V
	delete	DEL
	...	...

But what if there is more than one semantic link?
What if we want to add both hotkeys and popup descriptions to our actions?

```
Action CUT    = new Action("Cut",    Ctrl-X, "copy & delete");
Action COPY   = new Action("Copy",   Ctrl-C, "copy to clipboard");
Action PASTE  = new Action("Paste",  Ctrl-V, "paste from clipboard");
Action DELETE = new Action("Delete", DEL,    "delete selected text");
```

Do we introduce third list now? But what if we want to add one more property, or 17 more...
How do we handle situation with unknown and variable number of lists?

Don't fear, database experts solved this problem in the last millennium,
solution is called decomposition.

Decomposition

Let us identify variable parts in out previous code example.

```
Action CUT = new Action("Cut", Ctrl-X, "copy & delete");
```

As we know from before, CUT and Cut are syntax isomorphs so we can treat them as one entity, we will call this entity syntax identifier. The rest are semantically linked to syntax identifier so we will call them property1 and property2.

So we have following table:

Syntax Identifier	Property1	Property2
Cut	Ctrl-X	"copy & delete"
Copy	Ctrl-C	"copy to clipboard"
Paste	Ctrl-V	"paste from clipboard"
Delete	DEL	"delete selected text"

Now we need to realize that properties 1 and 2 are not linked in any way, not even semantically.

For example action descriptions might need to change at runtime due to localization support in our application and this change would not affect hotkeys.

We also might change hotkeys upon installation in order to support different hotkey standards on Windows, Mac and Linux but it would not affect description strings.

So we can break it down like this:

Syntax Identifier	Property1		Syntax Identifier	Property2
Cut	Ctrl-X		Cut	"copy & delete"
Copy	Ctrl-C		Copy	"copy to clipboard"
Paste	Ctrl-V		Paste	"paste from clipboard"
Delete	DEL		Delete	"delete selected text"

There are 2 ways to implement this decomposition in code, read on.

Partial decomposition

We started with this source:

```
Action CUT    = new Action("Cut",    Ctrl-X, "copy & delete");
Action COPY   = new Action("Copy",   Ctrl-C, "copy to clipboard");
Action PASTE  = new Action("Paste",  Ctrl-V, "paste from clipboard");
Action DELETE = new Action("Delete", DEL,    "delete selected text");
```

And decide to extract hotkeys is separate class:

```
const Hotkey HK_CUT    = Ctrl-X
const Hotkey HK_COPY   = Ctrl-C
const Hotkey HK_PASTE  = Ctrl-V
const Hotkey HK_DELETE = DEL
```

which leaves us with this modification of original code:

```
Action CUT    = new Action("Cut",    HK_CUT,    "copy & delete");
Action COPY   = new Action("Copy",   HK_COPY,   "copy to clipboard");
Action PASTE  = new Action("Paste",  HK_PASTE,  "paste from clipboard");
Action DELETE = new Action("Delete", HK_DELETE, "delete selected text");
```

Complete Docomposition

We extract messages too:

```
const String MSG_CUT    = "copy & delete";
const String MSG_COPY   = "copy to clipboard";
const String MSG_PASTE  = "paste from clipboard";
const String MSG_DELETE = "delete selected text";
```

and modify original accordingly:

```
Action CUT    = new Action("Cut",    HK_CUT,    MSG_CUT);
Action COPY   = new Action("Copy",   HK_COPY,   MSG_COPY);
Action PASTE  = new Action("Paste",  HK_PASTE,  MSG_PASTE);
Action DELETE = new Action("Delete", HK_DELETE, MSG_DELETE);
```

New code looks much more straightforward. It makes it impossible to make assignment errors, because you would see the error immediately:

```
Action PASTE  = new Action("Paste",  HK_DELETE,  MSG_PASTE);
```

It also enables you to modify hotkeys and strings independently, perhaps replace whole file/class depending on operating system, localization or whatever you need.

But wait say you, this is nothing but making a bunch of new constants, and it makes code a lot bigger. You are not actually making any new constants, you are just giving names to those that already exist, Ctrl-X and `"copy & delete"` are already there in your code.

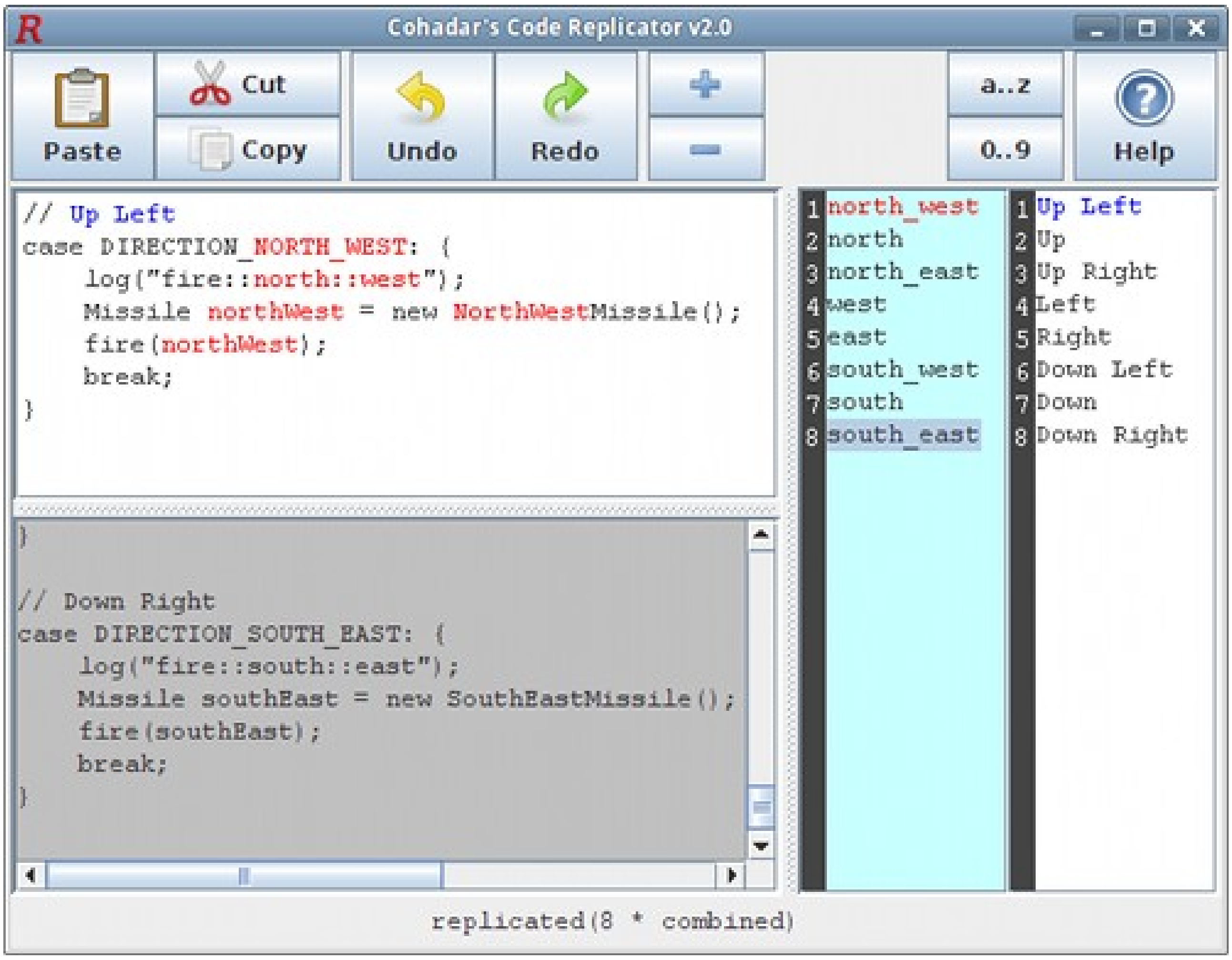
And code is much bigger only if you type it by hand, if you generate it you only need 2 lists and 3 template lines. Also note that you are not limited to one-line expressions. Once you extract away your semantic dependencies you will be able to generate functions, case statements or whole classes from a single list.

Conclusion

Embrace the power of syntax isomorphism.

Save yourself from repetitive typing and make your code more structured all at the same time:

http://codereplicator.sourceforge.net/



```
// Up Left
case DIRECTION_NORTH_WEST: {
    log("fire::north::west");
    Missile northWest = new NorthWestMissile();
    fire(northWest);
    break;
}
```

```
}

// Down Right
case DIRECTION_SOUTH_EAST: {
    log("fire::south::east");
    Missile southEast = new SouthEastMissile();
    fire(southEast);
    break;
}
```

1	north_west	Up Left
2	north	Up
3	north_east	Up Right
4	west	Left
5	east	Right
6	south_west	Down Left
7	south	Down
8	south_east	Down Right

```
replicated(8 * combined)
```