WIKIPEDIA

Intentional programming

In computer programming, **Intentional Programming** is a programming paradigm developed by Charles Simonyi that encodes in software source code the precise *intention* which programmers (or users) have in mind when conceiving their work. By using the appropriate level of abstraction at which the programmer is thinking, creating and maintaining computer programs become easier. By separating the concerns for intentions and how they are being operated upon, the software becomes more modular and allows for more reusable software code.

Intentional Programming was developed by former Microsoft chief architect Charles Simonyi, who led a team in Microsoft Research, which developed the paradigm and built an integrated development environment (IDE) called **IP** (for Intentional Programming) that demonstrated the paradigm. Microsoft decided not to productize the Intentional Programming paradigm, as in the early 2000s Microsoft was rolling out C# and .NET to counter Java adoption.[1] Charles Simonyi decided, with approval of Microsoft, to take his idea out from Microsoft and commercialize it himself. He founded the company Intentional Software to pursue this. Microsoft licensed the Intentional Programming patents Simonyi had acquired while at Microsoft, but no source code, to Intentional Software.

An overview of Intentional Programming as it was developed at Microsoft Research is given in Chapter 11 of the book *Generative Programming: Methods, Tools, and Applications*.[2]

Contents

Development cycle

As envisioned by Simonyi, developing a new application via the Intentional Programming paradigm proceeds as follows. A programmer builds a WYSIWYG-like environment supporting the schema and notation of business knowledge for a given problem domain (such as productivity applications or life insurance). Users then use this environment to capture their intentions, which are recorded at high

level of abstraction. The environment can operate on these intentions and assist the user to create semantically richer documents that can be processed and executed, similar to a spreadsheet. The recorded knowledge is executed by an evaluator or is compiled to generate the final program. Successive changes are done at the WYSIWYG level only. As opposed to word processors, spreadsheets or presentation software, an Intentional environment has more support for structure and semantics of the intentions to be expressed, and can create interactive documents that capture more richly what the user is trying to accomplish. A special case is when the content is program code, and the environment becomes an intelligent IDE.[3]

Separating source code storage and presentation

Key to the benefits of Intentional Programming is that domain code which capture the intentions are not stored in source code text files, but in a tree-based storage (could be binary or XML). Tight integration of the environment with the storage format brings some of the nicer features of database normalization to source code. Redundancy is eliminated by giving each definition a unique identity, and storing the name of variables and operators in exactly one place. This makes it easier to intrinsically distinguish declarations from references, and the environment can show them differently.

Whitespace in a program is also not stored as part of the source code, and each programmer working on a project can choose an indentation display of the source. More radical visualizations include showing statement lists as nested boxes, editing conditional expressions as logic gates, or re-rendering names in Chinese.

The system uses a normalized language for popular languages like C++ and Java, while letting users of the environment mix and match these with ideas from Eiffel and other languages. Often mentioned in the same context as language-oriented programming via domain-specific languages, and aspect-oriented programming, IP purports to provide some breakthroughs in generative programming. These techniques allow developers to extend the language environment to capture domain-specific constructs without investing in writing a full compiler and editor for any new languages.

Programming Example

A Java program that writes out the numbers from 1 to 10, using a curly bracket syntax, might look like this:

```
for (int i = 1; i <= 10; i++) {
   System.out.println("the number is " + i);
}
```

The code above contains a common construct of most programming languages, the bounded loop, in this case represented by the `for` construct. The code, when compiled, linked and run, will loop 10 times, incrementing the value of *i* each time after printing it out.

But this code does not capture the *intentions* of the programmer, namely to "print the numbers 1 to 10". In this simple case, a programmer asked to maintain the

code could likely figure out what it is intended to do, but it is not always so easy. Loops that extend across many lines, or pages, can become very difficult to understand, notably if the original programmer uses unclear labels. Traditionally the only way to indicate the intention of the code was to add source code comments, but often comments are not added, or are unclear, or drift out of sync with the source code they originally described.

In intentional programming systems the above loop could be represented, at some level, as something as obvious as "`print the numbers 1 to 10`". The system would then use the intentions to generate source code, likely something very similar to the code above. The key difference is that the intentional programming systems maintain the semantic level, which the source code lacks, and which can dramatically ease readability in larger programs.

Although most languages contain mechanisms for capturing certain kinds of abstraction, IP, like the Lisp family of languages, allows for the addition of entirely new mechanisms. Thus, if a developer started with a language like C, they would be able to extend the language with features such as those in C++ without waiting for the compiler developers to add them. By analogy, many more powerful expression mechanisms could be used by programmers than mere classes and procedures.

Identity

IP focuses on the concept of identity. Since most programming languages represent the source code as plain text, objects are defined by names, and their uniqueness has to be inferred by the compiler. For example, the same symbolic name may be used to name different variables, procedures, or even types. In code that spans several pages – or, for globally visible names, multiple files – it can become very difficult to tell what symbol refers to what actual object. If a name is changed, the code where it is used must carefully be examined.

By contrast, in an IP system, all definitions not only assign symbolic names, but also unique private identifiers to objects. This means that in the IP development environment, every reference to a variable or procedure is not just a name – it is a link to the original entity.

The major advantage of this is that if an entity is renamed, all of the references to it in the program remain valid (known as referential integrity). This also means that if the same name is used for unique definitions in different namespaces (such as "`.to_string()`"), references with the same name but different identity will not be renamed, as sometimes happens with search/replace in current editors. This feature also makes it easy to have multi-language versions of the program; it can have a set of English-language names for all the definitions as well as a set of Japanese-language names which can be swapped in at will.

Having a unique identity for every defined object in the program also makes it easy to perform automated refactoring tasks, as well as simplifying code check-ins in versioning systems. For example, in many current code collaboration systems (e.g. Git), when two programmers commit changes that conflict (i.e. if one programmer renames a function while another changes one of the lines in that function), the versioning system will think that one programmer created a new function while another modified an old function. In an IP versioning system, it will

know that one programmer merely changed a name while another changed the code.

Levels of detail

IP systems also offer several levels of detail, allowing the programmer to "zoom in" or out. In the example above, the programmer could zoom out to get a level that would say something like:

```
<<print the numbers 1 to 10>>
```

Thus IP systems are self-documenting to a large degree, allowing the programmer to keep a good high-level picture of the program as a whole.

Similar works

There are projects that exploit similar ideas to create code with higher level of abstraction. Among them are:

- Concept programming
- Language-oriented programming (LOP)
  - Domain-specific language (DSL)
- Program transformation
- Semantic-oriented programming (SOP)
- Literate programming
- Model-driven architecture (MDA)
- Software factory
- Metaprogramming
- Lisp (programming language)

See also

- Automatic programming
- Object database
- Programming by demonstration
- Artefaktur
- Semantic resolution tree
- Structure editor

References

1. "Simonyi explains, 'It was impractical, when Microsoft was making tremendous strides with .Net in the near term, to somehow send somebody out from the same organization who says, "This is not how you should do things--what if you did things in this other, more disruptive way?'" (Quote from "Anything You Can Do, I Can Do Meta" (https://www.technologyreview.com/2007/01/01/227178/anything-you-can-do-i-can-do-meta/), Tuesday, January 9, 2007, Scott Rosenberg,

Technology Review. Archived (https://archive.today/y7ZVR#selection-619.435-619.692) 20 September 2020 at archive.today)
2. *Generative Programming: Methods, Tools, and Applications*, by Krzysztof Czarnecki and Ulrich Eisenecker, Addison-Wesley, Reading, MA, USA, June 2000.
3. Scott Rosenberg: "Anything You Can Do, I Can Do Meta (https://www.technologyreview.com/2007/01/01/227178/anything-you-can-do-i-can-do-meta/)." *Technology Review*, January 8, 2007. Archived (https://archive.today/y7ZVR) 20 September 2020 at archive.today

External links

- Intentional Software (https://web.archive.org/web/20160115203115/http://www.intentsoft.com/) - Charles Simonyi's company
- *The Death Of Computer Languages, The Birth of Intentional Programming*, a technical report by Charles Simonyi (1995) (http://research.microsoft.com/pubs/69540/tr-95-52.doc)
- *Intentional Programming - Innovation in the Legacy Age*, a talk by Charles Simonyi (1996) (http://citeseer.ist.psu.edu/simonyi96intentional.html)
- Edge.org interview with Charles Simonyi (interviewer: John Brockman) (http://www.edge.org/digerati/simonyi/simonyi_p2.html)
- Language Workbenches: The Killer-App for Domain Specific Languages? (http://www.martinfowler.com/articles/languageWorkbench.html) - Martin Fowler's article on the general class of tools that Intentional Programming is an example of.
- "Anything You Can Do, I Can Do Meta" (http://www.technologyreview.com/Infotech/18047/) Tuesday, January 9, 2007, Scott Rosenberg, *Technology Review*
- Awaiting the Day When Everyone Writes Software (https://www.nytimes.com/2007/01/28/business/yourmoney/28slip.html?ex=1327640400&en=d2d098d765b27104&ei=5088&partner=rssnyt&emc=rs), *The New York Times*, 28 January 2007
- *Is programming a form of encryption?*, by Charles Simonyi (2005) (https://web.archive.org/web/20171019081819/http://www.intentsoft.com/dummy_post_1/)
- *Appropriate Levels of Abstraction*, by Charles Simonyi (2005) (https://web.archive.org/web/20171019085651/http://www.intentsoft.com/appropriate_lev-2/)
- *The information contents of programs*, by Charles Simonyi (2005) (https://web.archive.org/web/20171019090103/http://www.intentsoft.com/notations_and_p-2/)
- *Feature X Considered Harmful*, by Charles Simonyi (2005) (https://web.archive.org/web/20171019091206/http://www.intentsoft.com/feature_x_is_co/)
- *Notations and Programming Languages*, by Charles Simonyi (2005) (https://web.archive.org/web/20171019091833/http://www.intentsoft.com/notations_and_p/)
- *Personal Observations from a Developer*, by Mark Edel (2005) (https://web.archive.org/web/20171019073851/http://www.intentsoft.com/random_observat/)
- Microsoft Research's educational video introducing their Intentional Programming system (https://web.archive.org/web/20040701023940/http://www.cse.unsw.edu.au/~cs3141/ip.asf) (ASF format, circa 1998, 20 megabytes)