Personal programming and the object computer

Trygve M. H. Reenskaug¹

Received: 3 October 2019 / Revised: 14 November 2019 / Accepted: 18 November 2019 / Published online: 21 December 2019
© The Author(s) 2019

Abstract

My objective is to create an intuitive computer for laypeople who want to go beyond ready-made apps and create programs to control their electronic environment. I submit Loke, a new kind of computer that is a universe of objects and nothing but objects. I call it an *object computer*. Loke is implemented in Squeak, a variant of Smalltalk, and is an extensible, conceptual model for execution, inspection, and exploration. It was first used to demonstrate how Ellen, a novice, programs a smart alarm clock through a GUI adapted to her competence, needs, and preferences. Informal demonstrations indicated that laypeople immediately grasp the idea of communicating objects that represent real things in their environment. They also wanted to use it for their own purposes. They were creative in identifying personal opportunities for Loke and in sketching out their implementations. Interestingly, expert programmers who attended the demonstration did not see the point of Loke. I have completed the programming of Loke qua conceptual model. The model underpins its potential security and privacy and sustains its object and message models. The Loke qua programming environment is still in its infancy, and its inherent security and privacy properties are still not realized in practice. A future *Loke device* will be accessible from anywhere and embedded in its own hardware to achieve them. The Loke IDE rests on Data–Context–Interaction (DCI), a new programming paradigm that leads to readable code with a clear architecture. I submit Loke for the pleasure of personal programming.

Keywords Personal programming · Laypeople programming · IoT · Smart home · Industry 5.0 · Loke · Object computer · BabyIDE · MVC · DCI · Smalltalk · Squeak

1 Introduction

This article is dedicated to owners of an IoT, amateur programmers, popular science readers, and generally interested people. (Expert programmers may find papers and articles that are better tuned to their needs.) The article is an interim report from an ongoing project. The conceptual model and a proof-of-concept implementation are completed, but much work remains to make it into a product.

The *modern computer* celebrated its platinum jubilee on the 21st of June, 2018. Precisely 70 years earlier, the world's first programmer wrote the world's first program, stored it, and executed it in the world's first *electronic, digital, stored-program* computer at the Victoria University of Manchester,

England. The architecture of this computer affectionately known as *Baby* followed the then well-known von Neumann model (Fig. 1). *Baby* is morphing into billions of communicating computers, many of them connected into a single, global machine.¹

My goal is to empower laypeople to control their electronic environment with programs that they write themselves. While a professional programmer is a highly trained expert, the personal programmer writes programs for personal use and values simplicity and convenience over programming sophistication. There will be millions of personal programmers, and I single out Ellen as one of the novices. The challenge is to create a new way of programming that is both intuitive and easy to use while it provides the security and privacy required by Ellen. My answer is a *personal object computer* that I name *Loke* (pronounced “Loukeh”) after a powerful and somewhat deceitful god in Norse

Communicated by Professor Bernhard Rumpe.

✉ Trygve M. H. Reenskaug
trygver@ifi.uio.no
<http://folk.uio.no/trygver/>

¹ Department of Informatics, Universitetet i Oslo, Morgedalsvn. 5, 0378 Oslo, Norway

¹ https://www.ted.com/talks/kevin_kelly_on_the_next_5_000_days_of_the_web.

Fig. 1 The von Neumann model is a solitary system

mythology. He sometimes assists the gods and sometimes works against them.²

The *personal computer* was born on Monday, December 9, 1968, at the Fall Joint Computer Conference in San Francisco when Douglas Engelbart demonstrated a “tool for thought” and an “extension of the mind”³:

When it was all over, the crowd rose to their feet and cheered, spellbound, enthralled. Not only had Engelbart introduced the world to the notion of the computer as a personal assistant controlled by a mouse, keyboard and cursor, he’d shown them a graphical user interface which formed the basis of the ‘windows’ he’d been manipulating, hyperlinks and the concept of the networked online realm we know today as the Web.

One of the many descendants of this demo is Loke, the subject of this article. I propose that personal programmers are best served by a multimedia integrated development environment (IDE) such as the Loke IDE. The unifying concept is the object; objects represent everything of interest, and there is nothing but objects. The program model achieves a strong separation of concerns with DCI, a new programming paradigm. The Loke IDE uses text to specify algorithms and diagrams to define the structure of collaborating objects. MVC is the conceptual model behind the user interfaces in the IDE itself and, where applicable, in the personal programs created with it.

Loke builds on several concepts, some new and some modified versions of well-known ideas. I describe some of them here as an introduction to the rest of the article.

² Loke is sometimes called *Loki*. He can take on many different shapes according to whim. https://en.wikipedia.org/wiki/Loki#Scandinavian_folklore.

³ <https://nationalpost.com/entertainment/books/the-mother-of-all-demos-how-douglas-engelbarts-launch-of-the-pc-presaged-steve-jobs-and-everything-silicon-valley-an-excerpt-from-will-storrs-selfie>.

1.1 The stand-alone object

The notion of objects was introduced by Dahl and Nygaard [3]. Their objects formed part of the foundation for Alan Kay’s object orientation: “...its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network.”

Like a computer, an *object* is a logical entity that encapsulates state and behavior and has a globally unique identity.⁴ Object state is in the form of references to other objects. Object behavior is specified with *methods* that are different from the well-known procedures. The conventional “calling a procedure” is replaced by “sending a message to an object.” There is a significant semantic difference. A procedure call is short for the procedure body; it is a compression. A Loke message send is an abstraction; the method to be executed is unknown at compile time, and its correctness must be based on trust.

Received messages trigger object behavior. This behavior can cause a change in the state of the object and also make the object send messages to other objects. An object neither “knows” why a received message was sent nor does it “know” what the receiver of a message that it sends will do with it. The object is stand-alone and unaware of its environment.

In software engineering and computer science, abstraction is the process of removing physical, spatial, or temporal details or attributes in order to more closely attend to other details of interest.⁵ A common abstraction on objects in programming and computer science is the *class abstraction*. A multitude of languages such as Java and Ruby support it:

1. An object is an instance of a class.
2. The class discloses the inner construction of the object.
3. The instances of a class form an unordered set that hides the identity of its members.
4. An instance of a class has no information about its environment, i.e., its context.

1.2 The single, global machine as a universe of objects

Just like a single gear in a clockwork, an object by itself isn’t very interesting. The essence of object orientation is that objects collaborate to achieve a goal. The alternative *Role abstraction* describes an object in the context of its collaborators:

⁴ The identity is an immutable attribute associated with the object upon creation and is never shared by other objects after that.

⁵ [https://en.wikipedia.org/wiki/Abstraction_\(computer_science\)](https://en.wikipedia.org/wiki/Abstraction_(computer_science)).

1. An object is an entity that has an immutable and globally unique identity and that encapsulates state and behavior.
2. The object's encapsulation hides its inner construction from its outside.
3. The object provides a message interface that can be invoked by messages from other objects. The object may also send messages through *webhooks* (custom callbacks) to other objects.
4. The object plays a *Role* when it is collaborating with other objects.

The class and Role abstractions are complementary: The *class abstraction* exposes the object's state and behavior. Its identity is hidden in the unordered set of the instances of its class. The *Role abstraction* is the inverse. An object has an invariant and globally unique identity, while its state and behavior are hidden within its encapsulation.

The *raison d'être* for these abstractions are that they help the human mind create models of complex realities. All my researches have similar goals; it must lead to something useful to a human to be of interest. My style of research is experimental. When I get an idea, the next step is to express it in code for a reality check. In the early days with assembly and Fortran, I had to create a formwork in the shape of a design before pouring the code into it. Today, I have Smalltalk and go directly from idea to code.

There have been quite a few ideas through my 60 years of programming and software research. Most of them had a short life span ranging from hours to weeks and months. Some of them have survived to this day and form the foundation for Loke and this article.

I have created a laboratory for experimenting with objects and the object computer. I did consider to base it on a von Neumann machine, but I rejected this alternative because it lacks the essential peer-to-peer communication of modern computing. I decided to base my laboratory on the Squeak variant of Smalltalk because it is a universe of communicating objects, much like the single, global machine.

Figure 2 illustrates how I merge my Squeak⁶ universe of objects with objects found on the Net. The result is Loke, a *personal object computer*. Loke is personal and private in the sense that a smartphone is personal and private. It is an instance of a general platform that gets personalized when its owner augments it with personal data such as address books, bookmarks, letters, and personal programs.

The ground rules for the personal object computer are:⁷

Fig. 2 Ellen's Loke with her personal and shared objects

- Everything is represented by an object
- Objects have their own state (that is represented by objects)
- Objects communicate by sending and receiving messages (that are represented by objects).
- Objects have behaviors (methods, scripts, or server behavior) that define the meaning of received messages.

In a von Neumann computer, a *program* is a collection of instructions for performing a specific task. In the Role abstraction, a *program* is a collection of messages for achieving a specific goal.

1.3 The data–context–interaction (DCI) paradigm

In [17], I suggested that a program should be organized in a structure of communicating components where a component is an object that encapsulates other objects. The idea worked well for certain use cases, but it transpired that it was counterproductive in other cases. The problem was that the structure was static; one structure had to fit all. In DCI, the idea is the same, but the structure is dynamic; each use case has its own object structure that is built at runtime.

The *Data, Context, Interaction (DCI) programming paradigm* embraces the notion of programming: System state is declared as *Data* in the class abstraction. System behavior is declared as *Contexts* and *Interactions* in the Role abstraction:

- The **Data** are stand-alone objects such as the objects that represent things in Ellen's world.
- A **Context** is a structure of selected objects that play their Roles to meet Ellen's goal (a use case).

⁶ <https://squeak.org/>.

⁷ Inspired by [13].

Fig. 3 MVC bridges the gap between the computer and the human user

- An **Interaction** is a code within a Context that augments the behavior of the participating object to make things happen.

The DCI model of computing is on an abstraction level above the von Neumann model with its operating systems and programming languages and protects the personal programmer from their idiosyncrasies.

1.4 The model–view–controller (MVC) paradigm

The operators of the Manchester Baby computer worked it through a control panel like a TV remote. This panel has morphed into *graphical user interfaces (GUIs)*. The consequences are deeply radical for individuals and society alike. Figure 3 illustrates model–view–controller (MVC), a conceptual model of the GUI.⁸ The User experiences MVC as Douglas Engelbart’s “extension of their mind.” MVC uses two artifacts to achieve the “magic” of Fig. 3. First, the digital *Model* object shall faithfully represent the user’s mental model. Second, MVC uses well-known graphical representations in the *View objects* to present the information contained in the *Model* object in a way that is readily intuited by the *User*. An invisible *Controller object* sets up the Views in a window on the screen and coordinates them by making a selection show itself in all Views simultaneously. The Users no longer feel they are running von Neumann computers but achieve their goals by interacting directly with their mental model through the Views. A significant side effect of a successful GUI is that the underlying machine fades into the background in the user’s mind.

BabyIDE is a proof-of-concept implementation of Loke with an MVC architecture. The MVC Model is an abstract

Fig. 4 Engineering drawing of a solid in three projections

representation of a program. Views on the Model let the programmer work with the program seen in different projections. The Views are like engineering projections with their plan and side views of a solid (Fig. 4). Each View tells part of the story; all Views together tell the whole story.

Variants of BabyIDE are designed for different categories of users. Novice users can enjoy a low entry threshold with pre-defined Models and automatically generated Views. Expert users need to control all of M, V, and C. Both build on the same conceptual Model. The transition from novice to expert can be a smooth growth guided by the personal programmer’s increasing experience and needs.

1.5 The rest of the article

The rest of the article is organized into seven parts and three appendices:

Section 2: Novice Programming. Ellen, a novice Programmer, has a simple problem that she solves by editing a few simple Views on the program.

Section 3: Loke: The Personal Object Computer. I discuss the nature of Loke, its model, and its use.

Section 4: Further work. There is still a long way to go.

Section 5: Related work. Other initiatives that are related to personal programming.

Section 6: Summary and Conclusion

Section 7: Credits. Acknowledging sources of pictures, etc.

There are three appendices:

⁸ <http://folk.uio.no/trygver/themes/mvc/mvc-index.html>.

Appendix 1: ProkonPlan, an Example. (A comprehensive BabyIDE programming example with an architecture that combines MVC with DCI)

Appendix 2: BabyIDE, The Loke Integrated Development Environment (The LokeIDE user interface)

Appendix 3: BabyIDE, the first Loke implementation (Program documentation)

2 Novice programming

I recently observed a two-year-old girl, Selma, poking the TV screen and getting very upset because it did not answer her manipulations. (She plays with an iPad in the kindergarten.) It is a sign of the times that she learns to operate a computer before she learns to talk. I expect she will later find it natural to bend her devices to gratify her demands. *Personal Programming (PP) is what a person does to fulfill personal needs.* Some examples: The owner of a smart home will integrate its various things to create a Personal Home. A child will direct the behavior of their toys. A student will create a personal simulation program to understand a physical phenomenon better. A computational chemist will write a personal program that merges several chemical simulations into a coherent whole. An investor will create a personal program for working the share market. The human part of industry 5.0 can create personal programs to bridge the gap between man and machine. All of them want to take control of their environment with its devices and services: They need to become personal programmers.

I do not know how many personal programmers there will be in the world, but I choose to aim my design for some tens of millions of them. I single out Ellen to represent all novices. It is impracticable to give her formal training in a mainstream programming language, so I replace the von Neumann computer with *Loke*, a new and more intuitive alternative. *Loke* is a *personal object computer* that is on an abstraction level above von Neumann. *Loke* is a universe of objects and nothing but objects. Some of the objects are personal and private to Ellen, the rest she shares with others.

Ellen experiences her computer applications through their user interfaces, and the user interface *feels like* the program. I have experimented with an IDE for the Ellens of this world and call it *Loke/Novice*. I have three requirements:

1. *Loke/Novice* shall provide a low entry threshold by building on ideas that are well known to Ellen at the outset. The IDE shall protect her from the complexities of the hardware and its programming languages by working on a high abstraction level.
2. *Loke/Novice* shall scale from the first, small projects of the novice to the larger projects that Ellen can tackle as

she learns more to become an expert. The key is to build on the Loke model of computing throughout.

3. *Loke/Novice* lets Ellen employ a mixture of graphical and textual input to create her programs. The interface helps Ellen build a comprehensive mental model of computing by displaying the code that results from her gestures.

I have created a proof-of-concept implementation of Loke that embodies its conceptual model. This section describes how Ellen creates a simple and program with Loke's intuitive IDE. I use the terminology of the novice but aim its contents at the mentor who teaches her.

2.1 Ellen's smart alarm clock

Homeowners are in danger of being overwhelmed by disruptive smart home technologies. Ellen owns a smart home with an Internet of Things (IoT) that connects to her many things. Her IoT is again connected to the Web and thence to the whole Internet so that Ellen can access a vast number of resources that she can muster to serve her needs.

Ellen challenges us with her first, simple example. She plans to go on a long hike on the morrow, but only if it's going to be a dry day. Ellen needs to learn how to program an alarm clock that checks the weather forecast before it wakes her. The Danish philosopher Søren Kierkegaard has given this excellent advice:

If One Is Truly to Succeed in Leading a Person to a Specific Place, One Must First and Foremost Take Care to Find Him Where He is and Begin There.⁹

To rephrase Kierkegaard's question: "Where can I find the Ellens of this world?". I tried to find common ground for all humanity and looked up psychological research into the behavior of young babies. I didn't find any satisfactory answers. There was no consensus, and studies had only been on very small and select groups of infants. I lowered my goal and asked for some idea that is intuitive for a large and varied group of people. The idea of *composing* something from smaller parts was a good candidate. Most children have probably created a cow by sticking four pins into a piece of clay. The composition of LEGO® bricks into what Lego calls *projects* is an experience shared by many¹⁰ (Fig. 5):

Ellen's Lego construction toy provides her with a bucketful of bricks of different shapes and colors. She picks them

⁹ <https://teol.ku.dk/skc/sab/citater/>.

¹⁰ Lego products are sold in 130 countries. 75 billion LEGO bricks are sold per year, about 10 for each of the world population of 7.6 billion. On average, each person on Earth owns 75 Lego bricks. (<https://www.telegraph.co.uk/finance/newsbysector/retailandconsumer/8360246/Lego-in-numbers.html>).

Fig. 5 Kids compose Lego bricks into projects

Fig. 6 Ellen's IDE with resource objects and program composition

one by one and joins them to compose her project. Ellen composes her program in the same way. The Lego bricks become her resource objects that represent things in her world. She picks them one by one and joins them to compose her project, which is her program. (See an animation of the process here: <http://folk.uio.no/trygver/themes/Personal/PersonalProgramming.mp4>)

Ellen's box of objects appears as icons in the window called "Resources" in Ellen's IDE (Fig. 6, right). She selects one by one and drags it into her workspace in the window called "Context" (Fig. 6, left). She names it according to the Role it plays in her project and creates a road map by connecting it to other objects. The connections are client-server paths, as is assumed in most IoT communication standards. It is essential to understand that Ellen only works with objects. (Like everything else, collections of objects are objects too.) She doesn't know about classes and doesn't need them, so she leaves them to the experts.

Ellen first selects the smart loudspeaker that has replaced her old alarm clock on her bedside table. She picks it up, moves it into her project, and arbitrarily names it `WAKERUPPER` after the role it plays in her program. While a Lego brick

is a dead chunk of plastic, Ellen's objects are smart; they can do anything a computer can do. So she augments the `WAKERUPPER` role with a RoleScript that tells it what she wants it to do for her:¹¹

```
WAKERUPPER>>wakeMe
WAKERUPPER soundAlarm.
```

She tests it, and it works. Next, she needs a weather service, finds it in her box of objects, drags it in, names it, connects it up, and tells it what she wants it to do for her:¹²

¹¹ This is Smalltalk syntax, chosen for being more accessible to the novice. The Java equivalent is

```
void wakeMe() {WAKERUPPER.soundAlarm();}
```

¹² The Java equivalent is

Fig. 7 Ellen's user interface with examples of balloon help

Fig. 8 Ellen's DCI window for personal programming

```
FORECASTER>>checkWeather
FORECASTER expectedRainfall = 0
if True: [WAKERUPPER wakeMe]
```

Finally, she selects a timer that will perform her program in the morning:¹³

```
TIMER>>waitTillMorning
TIMER waitUntil: '06:00'.
FORECASTER checkWeather.
```

That's it, the program is complete, and Ellen sets her alarm clock by activating its first RoleScript.¹⁴

Footnote 12 (continued)

```
void checkWeather() {
  if (FORECASTER.expectedRainfall = 0) {
    WAKERUPPER.wakeMe();
  }
}
```

¹³ The Java equivalent is

```
void waitTillMorning() {
  TIMER.waitUntil ("06:00");
  forecaster.checkWeather()
}
```

¹⁴ My amateur video of the process is at <http://folk.uio.no/trygver/themes/Personal/PersonalProgramming.mp4>.

Fig. 9 Actors and interaction roles perform their RoleScripts

The idea of role scripts is new to Ellen, and she might find it hard to grasp the concept. I have chosen to give Ellen a menu-driven input interface in the form of a two-level menu of code templates (Fig. 7). It only requires her to be able to read the code, making it part of her passive vocabulary. A balloon text helps Ellen when the mouse hovers over a menu item.

A benefit of this form of visual programming is that Ellen can see the generated code and gradually include textual programming into her active vocabulary.

2.2 DCI is Ellen's mental model of computing

You may have noticed that Ellen's programming is far from conventional. Ellen's IDE brings modern human-computer interaction technologies to programming, and her IDE follows the MVC architectural pattern. The *Model* is invisible to Ellen; it consists of the fragments of code that, seen together during an execution, constitutes Ellen's program. Her three *Views* (Fig. 8) help her build her mental model while she uses them to create her alarm clock. An invisible Controller ties the Views together to create a coherent tool.

There is also a report generator that presents the program as a text.¹⁵

A theater analogy helps Ellen internalize the DCI programming paradigm: Data objects are like actors; they may be working or "resting." The *DCI Role* is like the theater's "function assumed or part played by a person or thing in a particular situation." The *DCI Context* is like a stage where actors perform their Roles. The *DCI Interaction* is like the scripts for the actors' parts:

Oxford English Dictionary **role**: "Early 17th century: from French rôle, from obsolete French roule 'roll', referring originally to the roll of paper on which the actor's part was written"

In DCI, the roll of paper is called a RoleScript (Fig. 9).¹⁶

Actors play their roles together with other actors on a stage; objects play their Roles together with other objects in a Context. Actors receive signals that cue their behavior; Roles receive messages that cue their scripts.

2.2.1 Ellen's data

In Introduction, I argued for basing Ellen's mental model of computing on the Loke computer. Its crucial ground rule was "*Everything is represented by an object*," which means that Ellen's Data are objects and nothing but objects.

Ellen's Data objects are objects in her Loke. Each object provides a self-explanatory message interface that she draws upon to compose her program. Some are general service objects; others represent familiar things in her smart home. Objects can also be instances of personal classes. She can write such classes when she becomes more proficient, or an expert can write them for her.

Ellen's program needs to answer three questions:

- What are the objects?
- How are they interconnected?
- What do they do?

The questions are open-ended, and many professional programmers find it hard to answer them. [20] discuss various techniques that help them find good answers. It is easier for Ellen because all her objects are pre-defined, and her goal is to fulfill her needs here and now. The point is that she perceived a need, poked around in her catalog of existing objects to select those that could help her, and then composed a program that served her needs. This work is much more concrete than if she had to start from scratch and had to invent the objects. Ellen is only interested in what the objects can do for her, i.e., their provided interfaces. It is irrelevant to her if they are implemented with classes or in some other way.

2.2.2 Ellen's context

Personal programs involve more than one object that performs their Roles within the DCI Context. In the interaction diagram (Fig. 8 upper left), an ellipse represents a Role.

¹⁵ Ellen's smart alarm clock code is at <http://folk.uio.no/trygver/assets/EllenCode-2019.11.05.zip>.

¹⁶ <https://en.oxforddictionaries.com/definition/us/role>

Fig. 10 Message sequence chart for Ellen's activateAlarm application

An arrow represents a link, i.e., a communication path for passing messages.

In a theater, the director casts actors to play the roles. Similarly, Ellen casts objects to play the Roles of her alarm clock by moving objects into her Context. Ellen can map any object to a Role as long as it has the capabilities needed by that Role. For example, any weather service can play the FORECASTER Role in Ellen's program as long as it handles the expectedRainfall message correctly.

Ellen's selected weather service plays a Role in her smart alarm clock. This weather service can, in its turn, implement the expectedRainfall operation in another Context within its information system. In general, any object can play a Role in a Context. It follows that since a DCI Context is represented by an object, it can play a Role in an outer Context recursively.

The DCI Contexts are recursive.

Ellen's programming tool, Loke/Novice, records Ellen's gestures by generating code for casting objects to Roles:

```

TIMER
  ^ResourceDictionary at: #clock
FORECASTER
  ^ResourceDictionary at: #weather
WAKERUPPER
  ^ResourceDictionary at: #speaker
  
```

As a novice, Ellen will not see this code. An expert Loke programmer, let's call him Anton, will use *Loke/expert* to replace it with more sophisticated methods for mapping objects to Roles.

2.3 Ellen's new way of programming

The programming tool that I have shown here is *Loke/Novice*, an experimental proof-of-concept implementation. I demonstrated the programming of a smart alarm clock to a farmer (my daughter). She had been using computers for many years but had never written a program. We were brainstorming some possible applications of Ellen's technology on her farm when she came with a pertinent observation: "This doesn't feel like programming at all." She was right, of course. Ellen's composition of her smart clock is closer to setting up a playlist in a music server than to programming in the conventional sense of the word. There is no self-contained source file and no discernible compilation stage. Instead, Ellen composed the required objects in her Context and wrote RoleScripts to control the flow of messages that meets her needs (Fig. 10).

The idea of RoleScripts is new to Ellen, and I endeavor to introduce her to it as smoothly as possible. I have chosen the Squeak default language for scripting because it is easy to read for the uninitiated.¹⁷ It is hard to read for most

¹⁷ You are free to define a different language for your BabyIDE RoleScripts if you know how to create a compiler for it in Squeak.

experienced programmers because it is different from what they are used to. The main stumbling block is the syntax for messages. A typical mainstream syntax for a procedure call is:

```
fileDirectory.copy (london, paris);
```

The Squeak syntax for a message send is more informative:

```
fileDirectory copyFrom: paris to: london.
```

I give priority to the uninitiated and hope experts reading this will have the patience to decode my simple examples. More information about the Squeak language is on the Squeak home page.¹⁸

LokeIDE is the programming interface for Loke. It caches Ellen's personal objects as well as some shared ones, and she adds more objects as needed. The addition extends the expressiveness of her language without changing her fundamental model of computing or even her IDE. I envisage that Ellen's mentors will make new objects with their message interfaces available to her as she needs them. When she gets more proficient, she may learn to do it herself.

Ellen's way of programming heralds a lifting of the universe of discourse from the hardware and the von Neumann machine to the human user and the universe of objects in Ellen's Loke computer. This fundamental advance is the theme of the next section.

3 Loke: the personal object computer

We are entering the connected society. Every thing, every person, and every Loke will be connected to a communication network. Some of these networks will be isolated; some will be connected through the Internet. An IoT is a network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and connectivity, which enables these things to exchange messages.¹⁹ The Internet, with its connected IoTs, can be seen as a *single, global machine* consisting of a universe of communicating objects. It is an artifact created by hardware and software and has no physical or logical center. This single, global machine is still in its infancy. Nobody quite knows what it will ultimately look like or exactly what it will be used for. And nobody knows if it sooner or

Fig. 11 Prokon's distributed personal computing (in figure, thick straight lines denote computer communication and thin lines indicate human communication. The figure is copied from [14])

Fig. 12 Humans coordinate their plans supported by their communicating Lokes

later will succumb under the increasing weight of its own complexity.

3.1 Personal distributed computing

The notion of personal distributed computing is not new. In the early 1970s, the *Prokon project* was part of a drive for participative work structures with decentralized command and control in business organizations [6, 14]. The idea was to mirror the organization's allocation of responsibility and competence in the distributed architecture of its information system. The immediate goal was to create a system that supported *decentralized planning while maintaining overall control*. The project introduced a vision of each manager owning a personal computer that they used for their personal tasks and for communicating with other managers (Fig. 11). Managers also delegated parts of their peer-to-peer communication to their computer: Anton asks questions, requests changes, or sends reports to other managers. When Kari's computer receives a request or report, she can accept or

¹⁸ <https://squeak.org/>

¹⁹ https://en.wikipedia.org/wiki/Internet_of_things.

Fig. 13 A world of Lokes

reject it automatically, or she can let it wait for personal intervention.

One of Anton's tasks is to create a plan for his department and to coordinate it with his colleague's plans. Different departments have different responsibilities: One can be responsible for designing a part, and another can be responsible for manufacturing it. They may make their plans in different ways, but both understand the logic of their personal planning program and ideally write it themselves.

From time to time, managers meet to synchronize their individual plans. Their personal computers are running below the table in Fig. 12 and communicate to support their owners who are negotiating personally above the table.

The Prokon project's main sponsor went bankrupt, and the project was abruptly terminated.

Concurrently with the Prokon project, the developments at Xerox PARC in the 1970s were on a different scale in creativeness, impact, and funding.²⁰ One of the prime movers in this development was Alan Kay with his Dynabook ideas [11, 12] and his concept of object orientation:

In computer terms, Smalltalk is a recursion on the notion of the computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network. Questions of concrete representation can thus be post-

poned almost indefinitely because we are mainly concerned that the computers behave appropriately, and are interested in particular strategies only if the results are off or come back too slowly.

Though it has noble ancestors indeed, Smalltalk's contribution is a new design paradigm—which I called object-oriented—for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion [13].

Loke builds on Prokon's open structure of people and collaborations together with Kay's object orientation. Loke's objects include both the Squeak universe of objects and the objects found on the Net. It supersedes the von Neumann computer with an object computer and makes it personal like a smartphone. The result is Loke: A personal object computer.

Squeak, a variant of Smalltalk, is a universe of objects, and it is always executing. It mimics the emerging single, global machine in that both are universes of communicating objects. Both are always running and obey programs that are fragmented among participating objects that are identified at runtime: There is no identifiable program in closed form. The main difference between the two is that the global machine is an open system in the sense that its objects have an independent existence and ownership even when they are not connected to a Net. In contrast, a Squeak universe of objects is closed in the sense that its objects have no existence outside a Squeak execution.

Conceptually, Smalltalk (and Squeak) started its execution some time in the distant past, and my instance is a fork

²⁰ <https://www.techspot.com/guides/477-xerox-parc-tech-contributions/>.

Fig. 14 The communicating roles in Loke are on a new abstraction level

of that execution. The execution is paused when I store my objects (my *image*) in a file and is resumed when I load that file. The image file may be copied, and a new fork of the execution starts when that copy is loaded. The consequence is that Anton has not only his personal copy of the programs but also the exclusive ownership of his personal objects, programs, and execution thread.

3.2 The Loke conceptual model and machine

Loke was introduced in Introduction as a personal computer that results from merging the shared objects of the single, global machine with the personal objects of a Smalltalk/Squeak fork. Figure 2 illustrates the merged objects of Loke; Fig. 13 shows a more detailed picture.

I have created Loke to be as intuitive as possible without loss of expressive power so that millions of users can use it without prior formal training in programming.

Loke is a universe of objects and nothing but objects. Every object has an immutable and globally unique identifier.

Every object is characterized by its RESTful²¹ message interface.

An object is either shared or personal.

The objects may have different owners, representations, and access restrictions.

The conventional way of programming with its *write code, compile, load, and run* is replaced by *select and modify Loke objects*. There is no source code in its conventional, closed form.

Ellen refines her understanding of her Loke by programming it. Just as a smartphone user accumulates apps, Ellen

accumulates a fund of personal and shared objects that cover her various domains of interest. It can be said that while a domain-specific language specializes a computer for a specific domain; Ellen's use of her Loke specializes it for her own domains, needs, and preferences.

3.3 DCI: The Loke programming paradigm

When Ellen programmed her smart alarm clock in Sect. 2.1, she started the clock with a menu command that sent a message to the `TIMER` Role in the Interaction. This message marked the transition from one frame of reference to another; From Squeak with its objects and methods and classes to DCI with its Contexts, Roles, and RoleScripts. Figure 14 illustrates how they form new levels of abstraction above Squeak. A Role can only send messages to itself, the Roles it is linked to, and the object it represents. Objects have no references to the Roles that play them. This restricted visibility is an essential feature of the DCI strong separation of concerns that shield Ellen from the idiosyncrasies of Squeak and the programming levels below it.

3.4 Reliability, security, privacy

There is an increasing concern about the reliability, security, and privacy for information systems in general and IoTs in particular [9]. My understanding of the multitude of challenges and solutions is almost nonexistent, but I can observe my own PC. My hardware is a von Neumann machine with its CPU and memory (Fig. 1). A program resides in the memory, and the CPU executes whatever it finds there; my PC is inherently unsafe. I use a popular operating system (OS) that has only two security levels: *User* and *System*. Various security measures wrap my von Neumann computer and OS with layers of protection. The protection can't be perfect since new security patches appear almost weekly. The weakness seems to be that intruders find cracks in the defenses and pollute the memory with malicious code that is faithfully executed by the CPU.

As an example, I once found that company X offered an exciting program for a free trial. During the downloading process, my OS asked: "*Do you permit X to access your system?*" I had to answer YES while wondering why X must be allowed to plant any malware into my system secretly and why a less comprehensive permission couldn't suffice. What I have is an intrinsic insecure CPU with a security-wise naïve OS on top of it. No wonder so many PCs (including mine) are infested with Trojans and other malware. (An alternative is to ignore the fascinating software offerings on the Web, of course.)

Loke goes to the root of the problem by superseding the van Neumann computer with the safer Object Computer alternative. Loke is designed to operate on an abstraction

²¹ https://en.wikipedia.org/wiki/Representational_state_transfer.

level immediately above the hardware, and ensembles of collaborating objects replace the OS. I defined the nature of the Loke object in Introduction: “*Like a computer, an object is an entity that encapsulates state and behavior and has a globally unique identity.*” The encapsulation is the key: It theoretically separates the outside of an object from its inside. The outside is a possibly ignorant or evil world that can only access the object through its provided message interface. The inside can handle each incoming message as an inner sequence of messages passing through an ensemble of participating objects. The encapsulation provides a security barrier in every object as long as the encapsulation is the only path from the outside to the inside. The recursion and protection end when an incoming message is handled in some other way.

Bran Selic commented upon the sentence in italics above:

We discussed this earlier: I don’t think you are making a good case for security in Loke, here’s why:

Most security breaches occur not through an API, but through backdoor access to the underlying machinery. For example, Anton could (maliciously or unintentionally) configure the resource dictionary to a malware device, which corrupts the Loke virtual machine.

This is because, in high-level software systems, such as Loke, there is always another implicit interface: the interface between the application abstraction (e.g., a Loke Object) and the underlying software (e.g., virtual machine) or execution hardware that makes it work. It is typically in this lower layer that security traps are planted.

Encapsulation only works for entities interacting with Loke objects within the Loke plane/abstraction level. Unfortunately, there is no encapsulation between a Loke object and its supporting hardware/software layers.

I am both naïve and ignorant in these matters, and Bran’s arguments are convincing. Yet, I can’t but wonder if there are limits to their validity. Currently, I use a computer with many installed applications that share much of their supporting hardware/software layers. I wonder if the threats would be the same if these applications were remote applications that used their separate supporting hardware/software layers? In other words: Can the addition of communication to our model of computing mitigate the problems? (I call it shifting from a CPU-centric to a communication-centric paradigm in Sect. 3.7.) I hope the answer is yes when I in Sect. 3.8 build a Loke computer as a kiosk that gives access to remote applications.

I have read somewhere that all developers of IoT-related programs should understand privacy, safety, and other issues. I believe the Ellens and Antons of this world will have more than enough with their own problems without

taking on issues that should have been resolved in the infrastructure. Nevertheless, Loke must be able to handle exceptions without confusing its owner. A simple solution is to tell the user in a more or less humorous form that something went wrong, abort the execution, and send an error report to the Loke implementer. So Ellen may wake up at 10 o’clock in the morning in beautiful sunshine with an error message: *I didn’t wake you because something went wrong.*

3.5 Programming by thinking

A prerequisite for program reliability is that the program does what it is supposed to do and nothing else. I wrote my first program in 1957 using a first-generation computer. My programs were only a few lines of binary code. Yet the programs were so complicated that I couldn’t wrap my mind around them: They were unreadable. I embarked on my first major programming project in 1960. My old way of “*programming by inspiration*” clearly had to be replaced by something better. As I couldn’t get a better mind, I had to look at my programming habits. I first considered “*programming by testing*” and wrote a short loop within a loop on the back of an envelope. Exhaustive testing of all possible executions would take a few hundred years, so I quickly abandoned this approach²² and ended up with “*programming by thinking*”:

“Any method that prevents a programmer from writing code is a good method”.²³

These considerations led to a simple architecture with a clear separation of concerns: A central database surrounded by application programs. Each application was further decomposed into small subroutines in a call tree. I could now wrap my mind around the whole and each of the parts separately. On this foundation, my team built Autokon, a 50.000 lines assembly program for the computer-aided design and manufacturing of ships. It was first deployed in 1963 and sold as a software product in 1965. The software was deployed on a multitude of different computers in most of the world’s major shipyards and had relatively few bugs. Through the years, the program was extended with many new application programs within the same architecture.

Much later, Ellen programs her smart alarm clock according to the DCI paradigm, a paradigm for “programming by thinking.” Her Data are stand-alone objects that were developed independently. The Contexts with their Roles,

²² In practice, testing can never be complete, and Edsger Dijkstra states the obvious in this famous quote from the 1969 NATO conference on software engineering: *Testing shows the presence, not the absence of bugs.* (Dijkstra [5]).

²³ This author.

RoleScripts, and Interactions are orthogonal to the Data. In his master thesis, Hector A. Valdecantos found that the DCI programming paradigm, with its separation of concerns, helps to make the code more comprehensible and thus appear simpler than comparable Java code [19]. Other examples of using DCI to improve simplicity are Bluemke [1] and a DCI-structured program for managing book loans in a library.²⁴

In his 1991 Turing Award Lecture, Tony Hoare succinctly stated the value of simplicity:

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” (Hoare [10])

3.6 Programming by collaboration

Peer review is a well-known technique for finding bugs early. I first read about it in a Byte Magazine of about 1969. The article argued for peer review of code as a very powerful method for removing deficiencies from a program while both the writers and reviewers of the code learn from the experience.²⁵ At the time of the Byte article, my team was creating a small program with a database-centric architecture and applications written in Fortran. The project progressed through weekly progress meetings, ensuring that the whole team assimilated the current version of the database schema and the program call tree.

The unit for review was the subroutine: The team assigned a subroutine to a team member who designed it, wrote the code, and passed it through the compiler to remove syntax errors. The routine then went back to the progress meeting that assigned it to another team member for review. The team tolerated and even expected that people make mistakes when coding. They also expected that the reviewer should find all of them. Hence, the reviewer was solely responsible for the correctness of the code, and the writer was out of the picture. No deficiencies were found in 3 out of 4 subroutines in unit testing. There were only minor bugs in the remaining routines. No defects were found during system testing or in the program’s lifetime.²⁶ The process, with its peer review,

was quite time-consuming with weekly progress meetings, reviews, and unit tests. Many projects do not have the time and human resources for such elaborate development processes, but they do find time for lengthy testing and rework.

We only did one experiment, and the program was very small. The experience was promising, and the team was motivated to try it out on a more extensive example. At the same time, we progressed from Fortran to object orientation. Code review was no longer feasible because we could no longer identify isolated chunks of code for review. Our only abstraction on objects was the class abstraction, and the only available chunks were class specifications. A class has dependencies in two dimensions: Up the class inheritance tree and across between the classes of collaborating instances. We could not find chunks of code that could be independently specified, written, reviewed, documented, and tested. We had to abandon peer review.

Software engineering is an exceptional branch of engineering. Engineering documentation is usually annotated with at least two signatures: *Created by (date and initials)* and *Checked by (date and initials)*. Object-oriented programs can have the first signature, but they are unreadable and can’t be checked. To quote the Design Patterns book [7] p. 22:

An object-oriented program’s runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. “The runtime structure consists of rapidly changing networks of communicating objects ..., it’s clear that code won’t reveal everything about how a system will work”.

The consequence of this chilling observation is that mainstream programmers are left with “Programming by testing” as the standard way of programming. Furthermore, their code doesn’t reveal everything about how their system will work.

Today, some 50 years after my first experience, code review is again feasible, and it would be interesting to use it for a real problem. We now have two abstractions on objects to identify chunks of code for review. The class abstraction gives chunks of code for the Data, i.e., stand-alone objects. The orthogonal Role abstraction gives independent chunks of code for the Role abstraction, i.e., system behavior. Peer review is again feasible. Dijkstra put it this way (Dijkstra [4]):

If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.

If we lift our eyes from coding to programming, our field of interest becomes much broader. Among other things, it includes the stakeholders:

²⁴ A library program: <https://github.com/ciscoheat/haxedci-example>.

²⁵ Later, Karl E. Wieggers cited inspections held on Motorola’s Iridium project that detected 80% of the defects present: http://www.processimpact.com/articles/seven_truths.html.

²⁶ We never aimed at testing quality into an inferior subroutine; the purpose of the testing was to confirm that the program was free from bugs. If a review found too many bugs in a subroutine, it should be scrapped and re-written by another team member. (This never happened in our small project.)

Fig. 15 The three paradigms of computing

A stakeholder is a person with an interest or concern in something especially a business. A stakeholder is someone who might be:

- Using a system
- Supporting a system
- Realizing benefits from a system.
- The organization or person paying for the system²⁷

This broad perspective on the process of programming is outside the scope of this article, and I refer interested readers to Coplien [2].

3.7 We need a paradigm shift

The history of Western astronomy shows a series of paradigm shifts from the geocentric paradigm with its stationary Earth as the center of the Universe with its essential complexities.²⁸ Astronomy evolved via the heliocentric to the current distributed paradigm with its chunks of mass connected by gravity. What appeared as essential complexity in one paradigm was easily resolved in the next.

It is tempting to look for similar paradigm shifts in computing. Mainstream programming has based much of its theory and practice on the CPU-centric paradigm exemplified

by the von Neumann machine. To me, a memory-centric paradigm came in 1960 with the Autokon central database [14]. The solution was obvious, and there must have been many similar initiatives without me being aware of them.

It is time to realize that the first two paradigms do not meet our current challenges: We are plagued with immensely large, complex, and insecure systems that long ago left the realm of human understanding. A recent example: Customers found that their bank charged them twice for the same transaction. Several weeks after the problem was discovered, the bank publicly admitted that they still didn't understand how the problem could arise: The complexity of their system was clearly beyond human comprehension. The bank has a staff of very competent experts, but they need a better foundation for modeling and implementing their sophisticated requirements.

Computers can transform, store, and communicate data (Fig. 15). The essence of the CPU-centric paradigm is that computers are primarily used to transform data; they *compute*. The essence of the memory-centric paradigm is that computers are primarily used to store data; they organize applications around a shared database. The essence of the communication-centric paradigm is that computers are primarily used to exchange messages with other computers to make them collaborate to achieve a common goal. It is worth noting that while the astronomers' new paradigms replace the old, the new computing paradigms extend the old as illustrated in the figure.

It is time to heed Tony Hoare's plea for simplicity and achieve a better way of separating concerns. Mainstream programming should embrace the *communication-centric*

²⁷ https://computersciencewiki.org/index.php/Planning_and_stakeholder_involvement.

²⁸ https://en.wikipedia.org/wiki/Geocentric_model.

paradigm as its fundamental model of what computing is all about. It is exemplified by the object computer that is the foundation for this article, and even MVC belongs here with its Model and the user interface as communicating entities.

The communication-centric paradigm has been on the horizon for many years. I first met it in Prokon's idea of distributed computers [15], but there must have been many other initiatives. A newer example is *Service-Oriented Architectures (SOA)* that, in essence, is communication-centric. It didn't meet with immediate success, possibly because people tried to apply it within the CPU-centric paradigm where it doesn't belong. There are many other examples such as distributed computing.²⁹ And of course, DCI and the IoT itself are, by definition, communication-centric.

3.8 The dream of a Loke computer

The original idea was that the Loke computer should be a piece of hardware that, like Alan Kay's Dynabook [11, 12], should cater to all its owner's computing needs. The computer was to be self-contained and includes the hardware and software needed to run a Loke personal computer. I have dropped this idea. Potential users like Ellen depend on her smartphone in her daily life. She uses it for many purposes ranging from paying for her groceries to finding the meaning of the word *astrobleme*. It is not practicable to reprogram all these services on a dedicated Loke device. The first Loke device must, therefore, be like an existing smart device augmented with Loke.

Ellen owns many devices, and she wants to run her Loke on all of them. Loke should, therefore, be deployed as a *remote application* on a suitable server:

A remote application is an application delivery solution wherein the actual application is installed on a central server and is used from a remote device. The end user receives screenshots of the application while being able to provide a keyboard, thumb tap, and mouse inputs. Remote apps have many names: remote application, server-client apps, app remoting, application virtualization, and virtual apps. The RDP protocol is one of the more popular protocols used to transmit data from the datacenter-hosted application to the remote devices.³⁰

Ellen may still own a Dynabook-like device and use it to access any available program on the net, including her personal Loke. The device will be like a program kiosk:

"Kiosk software is the system and user interface software designed for an interactive kiosk or Internet kiosk enclosing the system in a way that prevents user interaction and activities on the device outside the scope of execution of the software"³¹

Some of the resources on the Net are apps that are intended for human use, such as drawing programs, word processors, and Web browsers. Other resources are intended for machine-to-machine (M2M) use and can appear as Resource Icons on the Loke IDE desktop. Ellen can use her Loke to create and deploy new resource objects that are either personal or shared and that are intended for human or machine use.

A Loke device only needs a minimal operating system such as a microkernel³² or Squeak itself. This will significantly reduce its vulnerability to malicious attacks as will the security barriers between the kiosk and remote applications. The Loke device is a shell; everything happens somewhere else.

3.9 Teaching children computing

There is a widespread interest in teaching children about computers and programming, ranging from the voluntary "teach the kids coding" initiatives to its formal inclusion in school curricula. For example, the UK government has established a *National curriculum in England: computing programmes of study*³³ that specifies a mandatory course covering a child's 11 years of schooling.

Common to many of these initiatives is that the *solitaire computer* is at the center of attention; the programming of communicating machines is usually missing. It's as if the education of drivers should focus on the construction of the automobile without mentioning traffic.

I believe the initiatives start on the wrong foot by being unduly abstract and alien to the human mind. Compare with Ellen's *programming by composition*. Ellen need not follow a set curriculum, but she learns more when she needs more. She will progress from *Loke/Novice* to *Loke/Expert* as she gains experience with Loke programming. More important will be to integrate personal programming in general and Loke in particular with her different school subjects. Helped by her teacher, Ellen will use her Loke to build executable models of the subject matter, using available resource objects to fill in the details. A limiting factor to the teaching of conventional programming has been the lack of teachers

²⁹ https://en.wikipedia.org/wiki/Distributed_computing.

³⁰ <https://www.parallels.com/blogs/ras/remote-application-overview/>.

³¹ https://en.wikipedia.org/wiki/Kiosk_software#Security_features.

³² <https://en.wikipedia.org/wiki/Microkernel>.

³³ <https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study>.

with relevant competence. The low entry threshold of programming by composition significantly reduces this problem, and teachers can learn while they teach.

Ellen's programming of a smart alarm clock is a straightforward example that illustrates Loke's fundamental concepts in a simple presentation. At the other extreme is Hans Rosling's TED presentation: *The best stats you've ever seen*.³⁴ Rosling created this presentation by merging available world statistics with sophisticated graphics presentation tools. An expert like Anton should be able to compose a similar presentation from the same resources in his Loke, thus learning about the world through exploring official statistics.

Developers of school curricula will be offered a new dimension in teaching: Learning by modeling and exploring [11, 12].

4 Further work

I see no reason to modify Loke's conceptual model. Its current implementation, BabyIDE, has been "programmed by inspiration," and its architecture and code leave much to be desired. The current BabyIDE is an alpha version. It lacks several important features but suffices for a proof of concept as it supports the two essential Loke illusions:

- The Loke memory appears as a uniform universe of objects independently of the diversity of their locality, access mechanisms, and implementation details.
- All messages are handled the same way independently of the protocol used for message transmission and the nature of the receiver.

BabyIDE is theoretically secure against certain attacks, but it has to be re-implemented to carry this over to practice. The current implementation has the features needed for advanced Loke programming, such as re-implementing BabyIDE itself. The implementation has the hooks required for distributed programming, but this can't be demonstrated since BabyIDE is not yet connected to a Net.

I hope somebody will make a beta version available to anybody who has the courage and interest to be on the bleeding edge of the development. This version should be based on a re-implemented BabyIDE. Its opening screen will look something like Fig. 6, and the many Squeak facilities will be hidden in the manner of Etoys.³⁵ The re-implemented

BabyIDE will still run in my old version of Squeak³⁶ and must be ported to the latest version.

5 Related work

If everything I have learned through 60 years of programming were to be collected in a book, what I do not know would need a vast library. I am only too aware that I have missed essential initiatives and offer my sincere apologies for not having discovered them. Below is some related work that I have found.

5.1 Realizations of the DCI paradigm

5.1.1 Cope's trygve language

The DCI programming paradigm has been reified by James O.Coplien (Cope) in a research language he calls **trygve**:

- The **trygve** language is just one part of a system design that supports end-user mental models. In the end, **trygve**'s main contribution is to the left-brained side of computation—the enactment of scripts. Users still engage their right brain during program enactment, but in modern computing, such activity is usually associated with the visual cortex. Identifying the proper entities (objects) happens on the screen, and Model–View–Controller (MVC) has been designed as the bridge between the end user and computer in that regard. MVC and **trygve** can powerfully be combined to provide the most expressive links between the end user and the machine.³⁷

The source code of a **trygve** program has the following structure:

³⁴ <https://www.youtube.com/watch?v=hVimVzgtD6w>.

³⁵ <http://www.squeakland.org/>.

³⁶ Squeak version 3.10.2.

³⁷ <http://fulloo.info/Documents/trygve/trygve1.html>.

```

class className {class declaration}
...
context contextName {
.   role roleName {
.     roleScript header () {roleScript body} requires {role-player interface}
.     ...
.   }
.   ...
.   context instance variable declaration
.   ...
.   public contextName () {// the context constructor maps roleNames to role-players
.     roleName = expression;
.     ...}
.   public system-operation-name () {
.     system-operation script // this script handles and triggers a use case.
.   }
.   ...
}

```

The **trygve** syntax is somewhat like Java, but its semantics is based on DCI. It has two important keywords, **context** and **role**, that have their DCI meaning. The **trygve** source code is a text that declares the program as a whole. This makes it possible for the compiler to analyze the program and support its typing system.

5.2 Rune Funch's Marvin language

Marvin is a programming language that Rune designed with DCI in mind. It builds heavily on C#. The first versions of Marvin can be seen as extensions to C#, where versions in the road map will most likely reshape some of the fundamentals to provide even better support for DCI.³⁸

5.3 Other initiatives

5.3.1 ObjectTeams

Stephan Hermann's ObjectTeams³⁹ has a concept similar to the DCI Role, but they wrap the role-player in a separate object. This means that a role-playing object has at least two identities: The identity of the object itself and the identity of the wrapper. ObjectTeams thus violates the definition of an object in the Loke Role abstraction: "*An object is an entity with an immutable and globally unique identifier.*" The dual identity may cause object schizophrenia problems that are hard to track down.⁴⁰

³⁸ <http://fulloo.info/Examples/Marvin/Introduction/>.

³⁹ <http://www.eclipse.org/objectteams/>.

⁴⁰ James O. Coplien, 2014: *Why isn't it DCI if you use a wrapper object to represent the Role?* http://fulloo.info/doku.php?id=why_isn_t_it_dci_if_you_use_a_wrapper_object_to_represent_the_role.

5.3.2 Actor model

The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation. In response to a message that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through messaging (obviating lock-based synchronization).⁴¹

Many people find it hard to wrap their minds around parallel processes, and this reduces the applicability of the actor model for Ellen and Anton. Maybe the Actors Model will become the Turing machine of the communication-centric paradigm?

5.3.3 Extensions of mainstream languages

The DCI paradigm has been realized as constructs in many mainstream languages such as Ruby, C++, Scala, and Java.⁴² There are obvious advantages to using a mainstream language. A disadvantage is that the DCI concepts are not part of the language, and its advanced features have to be stressed to the utmost. The result is that the code isn't as clearly organized as the BabyIDE code. Nevertheless, the application of DCI with some of these languages has shown improved program architectures compared to the conventional alternatives.

⁴¹ https://en.wikipedia.org/wiki/Actor_model.

⁴² <http://fulloo.info/Examples/>.

Fig. 16 UML diagram of a This Then That applet

5.3.4 Interplanetary file system (IPFS)

The Interplanetary File System (IPFS). It's a peer-to-peer (p2p) file-sharing system that aims to change the way information fundamentally is distributed across & beyond the globe. IPFS consists of several innovations in communication protocols and distributed systems that have been combined to produce a file system like no other. So to understand the full breadth and depth of what IPFS is trying to achieve, it's important to understand the tech breakthroughs that make it possible.⁴³

Loke may use something like this to store Ellen's personal data.

5.3.5 Microservices

Microservices are a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained, and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. It makes the application easier to understand, develop, test, and become more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous refactoring. Microservice-based architectures enable continuous delivery and deployment.⁴⁴

Microservices appear to be like primitive objects, and their fine grain and lightweight protocols could make them useful as resource objects in Loke.

5.3.6 Trusted platform module

In Sect. 3.8, I assumed that a Loke device would need hardware support to achieve the required security and privacy. The Trusted Platform Module seems to provide what's needed:

- Trusted Platform Module (TPM, also known as ISO/IEC 11889) is an international standard for a secure crypto-processor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys.
- The primary scope of TPM is to assure the integrity of a platform. In this context, “integrity” means “behave as intended,” and a “platform” is any computer device regardless of its operating system. It is to ensure that the boot process starts from a trusted combination of hardware and software and continues until the operating system has fully booted and applications are running.
- Discrete TPMs are dedicated chips that implement TPM functionality in their own tamper-resistant semiconductor package. They are theoretically the most secure type of TPM because the routines implemented in hardware should be more resistant to bugs versus routines implemented in software, and their packages are required to implement some tamper resistance.
- An example of TPM use for platform integrity is the Trusted Execution Technology (TXT), which creates a chain of trust. It could remotely attest that a computer is using the specified hardware and software.⁴⁵

5.3.7 If this then that (IFTTT)

IFTTT is an early commercial product in the emerging era of communication-centric programming:⁴⁶ “IFTTT is a free platform that helps you do more with all your apps and devices.”

Figure 16 shows the architecture of an applet, an IFTTT program fragment. The boxes represent resource objects called services. They come in two flavors: triggers and actions. A trigger service creates webhooks (custom callbacks) in response to some outside events such as the arrival of a new book in a library or a weather service observing the onset of rain.

An action service is an object that, like a resource object, responds to the messages in its provided interface. Some services can act as both triggers and actions. To create an applet, the programmer selects a trigger service and links it to an action service, as shown in the figure.

⁴³ <https://hackernoon.com/a-beginners-guide-to-ipfs-20673fedd3f>.

⁴⁴ <https://en.wikipedia.org/wiki/Microservices>.

⁴⁵ https://en.wikipedia.org/wiki/Trusted_Platform_Module.

⁴⁶ <https://ifttt.com/>.

The IFTTT product is owned and marketed by IFTTT Inc. Its main contribution to the realization of the object computer is the number of vendors who have made their products usable as services in the IFTTT environment. IFTTT Inc. claims to have connected IFTTT to more than 600 services including Amazon Alexa, Facebook, Instagram, Twitter, Fitbit, Slack, and more.⁴⁷ It is not clear to me if these adaptations are proprietary or if they are open and available to Loke

IFTTT is conceptually simple and is easy to learn for its target group. Currently, I have only seen the one programming construct; there is apparently no sequence and no loop. As I see it, IFTTT gives an early glimpse of programming under the communication-centric paradigm, but there is much more to come.

5.3.8 Industry 5.0

Industry 5.0 is a term used to denote the current stage in a long string of industrial production technologies. It started with what's now called Industry 1.0; the technologies made possible by the first industrial revolution. We have now reached a next step that includes the human as an essential part of industrial production. This introduction to *Industry 5.0*, a conference organized by the European Economic and Social Committee,⁴⁸ says what it is all about:

The fourth industrial revolution, or Industry 4.0, is characterized by the integration of smart, connected, and autonomous digital and physical technologies like IoT and robotics. While this presents a range of benefits and opportunities, studies show that many remain worried of robots taking their jobs. This is why this conference on Industry 5.0 is relevant as it can shed light on how we can better integrate people into a society based more and more on digital technologies.

While some see Industry 4.0 as wasting human problem-solving skills, value-adding human creativity, and the critical and exclusively human ability to deeply understand customers, Industry 5.0 is focused on combining human beings' creativity and craftsmanship with the speed, productivity and consistency of robots. Industry 5.0 means to better appreciate the cooperation between robotics and human beings by combining their diverging strengths, in order to create a more inclusive and human-centred future.

This conference will feature two discussion panels that will touch upon topics such as:

The technology behind industry 5.0, which can help human beings and robotics to work better together. This entails learning more about collaborative robotics and human-machine interface.

Most of the robots will be probably be delivered as RESTful servers or similar, while some may provide special-purpose machine-to-machine (M2M) languages for robotic collaboration. It seems to me that the concept of *the object computer* can tie the robotics and the human participants together in a common frame of reference. Further, the human participants can configure and program their system using personal programming.

6 Summary and conclusion

This article is about people and their objects. The reasons are twofold. First, my research has always been aimed at the human users of my results. Some of them are programmers, and some are from other walks of life. Second, my *mental frame of reference*⁴⁹ for computing is based on objects: System memory is distributed among objects, and system behavior takes the form of messages flowing between them. The word "object" occurs more than 300 times in the article's ~20 pages, and the word "subclass" no times.

A mental frame of reference is deep-seated, usually subconscious, and very hard to change. I have only changed my computing frame of reference twice during my 60 years of programming. The first was in 1968 when Edsger Dijkstra "considered the *goto*-statement harmful." To me, with ten years of professional programming behind me, it sounded like pure nonsense. But I looked back on my code and found it confusing and unreadable. So maybe Dijkstra had a point? (He usually had.) It took me six months of hard and very traumatic work to make the transition. One day my brain suddenly clicked over. Before the click, a program was a sequence of instructions. Suddenly, a program became a structure of modules, and the old frame of reference was gone. I no longer needed the *goto* and have never missed it.

The second change came gradually and without me being aware of it. It was triggered by my end users needing distributed personal computers in the Prokon project. The object-based frame of reference has grown unobtrusively in my mind, but I haven't expressed it explicitly until this article. A system is a universe of objects and nothing but objects. System memory is distributed among the objects. System behavior is a flow of messages through the objects. I found it

⁴⁷ <https://ifttt.com/services>.

⁴⁸ <https://www.eesc.europa.eu/en/agenda/our-events/events/industry-50>.

⁴⁹ A frame of reference is a complex set of assumptions and attitudes which we use to filter perceptions to create meaning. The frame can include beliefs, schemas, preferences, values, culture, and other ways in which we bias our understanding and judgment. (Changingminds).

implemented in Smalltalk when I came to Xerox PARC as a visiting scientist in 1978. Smalltalk is a universe of objects, an *image*. System state is distributed among the objects; system behavior takes the form of messages flowing from object to object. The first concrete outcome of my object thinking was MVC (“so obvious that it wasn’t worth writing about”). Later came role modeling, DCI, and Loke.

It came as a complete surprise to me that to the majority of the folks at PARC, object orientation was about classes, not objects. An example is in the CRC cards, a popular tool for brainstorming and teaching program design. The cards (usually index cards) are divided into three fields: *Class*, *Responsibilities*, and *Collaborators*. The tool’s fundamental flaw is its confusion of class and instance: *Classes do not collaborate, their instances do*. I hear people say: “*This class sends a message to that class.*” It makes me shudder. It is fundamentally wrong and may block the speaker’s chance of ever internalizing an object-based frame of reference.

My hard-earned experience with the difficulty of changing my own frame of reference is the motivation of the second sentence in this article: (*Expert programmers may find papers and articles that are better tuned to their needs.*) I have aimed the article at laypeople who have no ingrained frame of reference for programming.

I have the audacity to claim that a mental frame of reference based on objects is closer to a common human mindset than a CPU-based frame can be. I suggest you first look at the von Neumann model in Fig. 1 and observe how it invites the notion of a program being an abstract sequence of instructions. Then, look at Ellen’s programming interface in Fig. 6 and see that Ellen can construe its symbols as concrete components in her program composition. (I have no illustration of a class hierarchy, but hope you see the point.)

Loke’s conceptual foundation was briefly summarized in Introduction and was discussed in detail in the body of the article. It is all very simple and almost obvious to a mind with an object-based frame of reference:

- **The object** is an entity that encapsulates state and behavior. The encapsulation separates the object into an outside and an inside. Its outside is visible to its environment and is known by its immutable and globally unique identity. It can respond to a set of messages called its provided interface. It can also send messages to objects in its environment through their required interfaces. These interfaces are the glue that makes it possible to combine them into an *object computer*, or what Kelly called the great global machine. The inside of the object, hidden behind the encapsulation barrier, contains the implementation of the behavior triggered by incoming messages. The implementation may be realized by classes, by Fortran and assembly programs, or by huge systems like the meteorologists’ weather forecasting models. No wonder

classes are of secondary importance to Ellen. She will only need them if she wants to create her own resource objects. Most of the objects are, of course, found on the Web or provided by the vendors of her smart things.

- **Model–View–Controller (MVC)** mandates a simple structure of objects that bridge the gap between the human mind and the computer. The Model is an object or structure of objects that faithfully reflects Ellen’s mental model of the subject matter. One or more Views transform the abstract Model to a form that Ellen can intuit; this usually means some well-known presentation like a diagram or a text. (Years ago, I created a movie clip View that showed the message flow during an execution, but never an audio View.) The Controller sets up the Views and coordinates shared functionality like selection.
- **Data–Context–Interaction (DCI)** is a new paradigm for the separation of concerns in computer programming. The *Data* are quite simply stand-alone objects. The *Context*⁵⁰ is an artifact where Ellen musters the objects she needs to realize a use case. She composes them into a structure of communicating objects and names them according to the role they will be playing. The *Interaction* augments the participating objects with scripts that control their collaboration.

During the development of this article, the critical importance of privacy and security in our society has grown exponentially. Evil penetrators badly hit even the most sophisticated of organizations. It appears to be a never-ending war between good and evil, and the evil seems to be winning. I won’t even dream of taking on this problem that is challenging the best brains in our profession. But Ellen is my responsibility, and I might be able to help her. Perhaps, I can build on the work of the experts and create a Loke device that is exceptionally resistant to attacks. I speculated on ways and means in Sect. 3.8. The main security barrier is the encapsulation barrier between the outside and inside of an object: An evil outside can only reach the inside through its message interface, always assuming that there are no security holes in a possibly shared supporting software. I suggested a distributed solution so that Loke could run on its private copy of the supporting software. Ellen’s personal Loke became a kiosk, and I pushed the responsibility for its security and privacy onto the service providers. I’m an ignoramus when it comes to security and privacy but would like to explore if a distributed object model may help mitigate the challenges.

I wrote in Introduction that Loke was an ongoing project. It remains to report where it stands at the time of writing. I have built BabyIDE-1, a proof-of-concept implementation

⁵⁰ It’s called a *Collaboration* in UML and a *role model* in OOram (Reenskaug [16]) its predecessor.

of Loke. Embedded in this implementation is a conceptual model of Loke for exploration and experimentation. I regard Loke qua conceptual model as completed. The model underpins its inherent security and privacy and sustains its object and message models.

Loke qua a computer program leaves much to be desired. It has been “programmed by inspiration” and has served as my software laboratory. The conceptual model and Ellen’s IDE have been shaped through experimental programming in this laboratory. This implementation is adequate for a demonstration of simple personal programming: “*Ellen wants to create a smart alarm clock, and she programs it through a GUI that is adapted to her competence, needs, and preferences.*” The implementation has been used to demonstrate how Ellen, a novice, programs her smart alarm clock. Half a dozen informal demonstrations to laypeople indicated that they immediately grasped the idea of objects representing tangible things in their environment. They were creative in identifying personal opportunities for Loke and sketching out the flow of messages that could implement them. Interestingly, two trained programmers watching the demonstration appeared to be well served with their current technology and did not see the point of Loke.

Loke qua programming environment is still in its infancy: It is not connected to a net, and most notably, its inherent security and privacy properties have not been realized in practice. A future *Loke machine* embedded in its own hardware can achieve Loke’s security and privacy and will be connected to the Net in a client–server architecture.

The development of Loke has been a one-man project since I started it in 2015 as a continuation of the work reported in [17]. An advantage of being alone was that it was easy to drop one line of research at any time and start afresh on a new one (a frequent occurrence). The disadvantage was that it is limited to what a single person can do.

I am a nonagenarian and lack the necessary time and energy for bringing Loke to fruition. The time is ripe for scaling up the project with more people and better funding. So I’m searching for an innovator who will identify with the goal and realize its potential.

In the late 1970s, I implemented the first program with an MVC architecture. It later transpired that this implementation could be seen as a conceptual model of the human use of computers. Maybe that in the future, it will transpire that Loke can be seen as a conceptual model of the human use of connected things.

7 Credits

Several figures contain material downloaded from the Web and that is reproduced with permission.

Many of the images have been downloaded from Pixabay and are marked as such.

No attribution required for Pixabay pictures, but some is nevertheless added below.

Simplified Pixabay License is found at: <https://pixabay.com/no/service/license/>

Figure 17: The von Neumann model is a solitary system.

https://en.wikipedia.org/wiki/Von_Neumann_architecture
Kapooht [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)]

Figure 18: Ellen’s Loke with her personal and shared objects.

Picture: Simplified Pixabay License

<https://pixabay.com/photos/tablet-living-room-dog-woman-girl-843798/>

Picture taken by PourquoiPas from Pixabay

Figure 19: Engineering drawing of a solid in three projections.

Picture ©Emok 2008,

<https://commons.wikimedia.org/w/index.php?curid=4280545> reproduced by permission.

Figure 20: Kids compose Lego bricks into projects.

Left picture

<https://pixabay.com/no/photos/lego-spiller-bygge-modul-fargerike-1629073/>

Picture taken by Markus Spiske from Pixabay

Right picture:

<https://pixabay.com/no/photos/works-lego-robotikk-3492099/>

Picture taken by suchocka from Pixabay

Figure 21: Actors and Interaction Roles perform their RoleScripts.

Background picture:

<https://pixabay.com/illustrations/theater-stage-curtain-acting-399972>

Picture taken by Gerd Altmann from Pixabay

Foreground picture: (used twice):

<https://pixabay.com/illustrations/wizard-magic-sroll-mage-magician-1456914/>

Picture taken by GraphicMama team from Pixabay

Figure 22: A world of Lokes

left picture:

<https://pixabay.com/photos/child-tablet-technology-computer-1183465/>

Picture taken by Nadine Doerlé from Pixabay

middle picture:

<https://pixabay.com/photos/tablet-living-room-dog-woman-girl-843798/>

Picture taken by PourquoiPas from Pixabay

right picture:

<https://pixabay.com/vectors/elderly-wrinkled-man-old-aged-152866/>

Fig. 17 ProkonPlan user interface (The horizontal axis is marked in week numbers in panes 4 and 5.)

Fig. 18 BabyIDE for ProkonPlan screen dump (The BB9 prefix is an ugly attempt at overcoming the lack of namespaces in Squeak.)

Picture taken by OpenClipart-Vectors from Pixabay
Figure 23: The communicating Roles in Loke are on a new abstraction level.
inset picture:

<https://pixabay.com/photos/tablet-living-room-dog-woman-girl-843798/>

Picture taken by PourquoiPas from Pixabay

Fig. 19 BabyIDE window

Acknowledgments The work that has led to the DCI paradigm, BabyIDE, and Personal Programming has taken many years of ups and downs. I could not have stayed the distance if had not been for the encouragement I received from men I deeply respect, the foremost being Dave Thomas and Bran Selic. I have long known James O. Coplien (Cope) and thank him for many rewarding discussions over the years. Our common ground has been our focus on people. The value of a system is its value for its users. Users can be the end users of an application or its developers using a programming environment. We have both been following our separate paths when searching for a common truth we both have felt must be out there somewhere. In August 2008, I launched a new programming paradigm I called DCI—Data, Context, and Interaction. It was accompanied by BabyIDE, an Interactive Development Environment written in Squeak. I spread the good news to a very large number of people over the Web. There was no response except one: Cope wrote: “*Til lykke, Trygve. It’s not often that one can claim two great life accomplishments in one life.*” At long last, we joined forces to further the DCI ideas. I am grateful for Cope’s invaluable contributions to this work. Also, for the dissemination of the DCI paradigm which could not have happened without him. I sincerely thank also the active community centered around the object-composition@googlegroups.com mailing list for their contributions to the DCI paradigm and its dissemination. The first BabyIDE RoleScripts were implemented using Traits.⁵¹ I sincerely thank Nathanael Schärli, Stéphane Ducasse, Andrew Black, and Adrian Lienhard for providing this very powerful extension of the Squeak class paradigm that paved the way to augmenting Roles with behavior. The use of Traits was a crucial step toward the current strong separation of concerns between classes and Contexts. The concepts of Personal Programming and Ellen’s Loke Computer stem in part from Smalltalk that was created at Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, and the Learning Research Group [8]. The value of their disruptive ideas cannot be overestimated. Finally, my sincere thanks are due to Bran Selic for his thoughtful and always constructive reviews of a series of drafts of this article.

⁵¹ [https://en.wikipedia.org/wiki/Trait_\(computer_programming\)](https://en.wikipedia.org/wiki/Trait_(computer_programming)).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Appendix 1: ProkonPlan, an example

Anton is the manager of a design department. His plans are in the form of activity networks where an activity represents a task that needs to be done. An activity has a specified duration, it cannot start before all its predecessor activities are completed, and it must end before any of its successor activities can start. Other managers are responsible for other activities. Anton works with a personal fragment of the overall plan. His peers publish activity objects that Anton connects to, and Anton publishes activity objects that *they* can connect to.

Prokon/Plan is an activity network planning application that Anton can use for his personal fragment. Even if this is a centralized demo program without connection to other fragments, it illustrates the kind of program that Anton could use in a distributed environment. The application’s user interface has five panes, as shown in Fig. 17:

1. Identifies the program version and offers a menu for triggering various operations on the model (use cases).
2. Shows the properties of the selected activity, UI-Ctx, and allows them to be edited. (Notice that selections are synchronized; activity UI-Ctx is selected in all views simultaneously.)

3. Shows the activity dependency graph that connects predecessor and successor activities.
4. Shows the earliest and latest start and finish times. They are computed from the project's start and finish times, and activity duration and dependencies. The computation is invoked with a menu command in pane 1. The top, yellow bars show the earliest, and the bottom, green bars show the latest times.
5. Shows the results of an automatic resource allocation.

I used BabyIDE/expert to program Prokon/Plan as an example of MVC/DCI programming. This is a typical MVC application where the Model represents the user's mental model of the plan, and the Views are what the user works with. The M, V, and C parts are further separated into state and behavior by applying DCI to each of them, giving 6 projections. Their names are shown in the top line of the screen dump in Fig. 18. *Model-Context* is selected. The interaction diagram shows the participating objects, and the source pane shows the RoleScript for frontloading one activity at the time.

- *Model-data projection*. The user experiences the program's user interface as an extension of their mind. This "magic" of MVC (Fig. 3) is achieved by faithfully reflecting the user's mental model in the *Model-Data*. The classes are *DBase*, *Model*, *Activity*, *Dependency*, and *Resource*.
- *Model-context projection*. This projection implements Model behavior, i.e., the operations on the Model itself. This projection realizes the user's mental model of the planning operations. The Contexts are *BackloadCtx*, *FrontloadCtx*, and *ResourceAllocationCtx*.
- *View-data projection*. The "magic" of MVC is partially achieved by using well-known graphic languages for the Views so that the user readily intuits them. The views thus bridge the gap between the human mind and the Model. The classes are *DependencyView*, *ActivitySymbol*, *DependencyLine*, *ActivityTextView*, and *BlurbView*.

- *View-context projection*. The behavior of a View is to retrieve data from the Model and use the information to paint the View. There are many intricate coordinate calculations involved, and it enhances readability to separate them out in their respective Contexts: *DependencyDisplayCtx*, *GanttDisplayCtx*, and *ResourceDisplayCtx*. There are also Contexts for user input: *AddActivityCtx* and *AddDependencyCtx*; they trigger their respective *Model-Context* RoleScripts.
- *Controller-data projection*. The Controller creates the ProkonPlan window in Fig. 17 with its Views, and it is responsible for the selection mechanism. I only use DCI when it leads to more straightforward and readable code. The Controller is here regular class-oriented Squeak code because it seemed to be an overkill to separate state and behavior here. Consequently, there is no *Controller-Context* projection. There is one class: *Controller*.

A report generator has printed the ProkonPlan program on a file.⁵² The report clearly illustrates how the MVC/DCI separation of concerns leads to almost independent projections where each projection is separately written, reviewed, and tested. The exception is the selection mechanism that goes across many projections and is implemented in class-oriented code.

The loop for traversing all the activities is the responsibility of the *PLAN* Role (Fig. 18):

```
PLAN>>frontload
  PLAN allActivities do: [:act | act earlyStart: nil]. "set to unplanned"
  [ACTIVITY notNil]
    whileTrue:
      [ACTIVITY frontload.
       CURRENTCONTEXT remap.
      ].
```

The *CURRENTCONTEXT* role is responsible for mapping Roles to objects making sure that it selects an activity that is ready for planning:

```
BB9FrontloadCtx>>remap
  ACTIVITY := model allActivities
  detect:
    [:act |
     act earlyStart isNil and:
      [(model predecessorsOf: act) noneSatisfy: [:pred | pred earlyFinish isNil]]]
    ifNone: [nil].
  super remap "map remaining roles".
```

⁵² <http://folk.uio.no/trygver/assets/BB9Planning/>.

Fig. 20 The data class browser

Fig. 21 The context class browser. Note that an instance of a Context class (a Maestro) shall be a subclass of Resource to enable it to play a Role in an outer context. The figure shows that this feature is not yet implemented: Currently, (BB1) Context is a subclass of Object instead of Resource

A final remark:

In Fig. 17, pane 5, we see that the UI-Ctx activity is to be performed by Joe in weeks 2 and 3 and finished by Ava in week 4. This is probably not the desired allocation but rather a side effect of the default ResourceAllocationCtx algorithm. While simple algorithms compute front- and back-loading, the optimal allocation of resources depends on local circumstances. I suggest that managers should personally tune this algorithm to get an acceptable plan. This

is feasible because the script is wholly contained within a single Context, the ResourceAllocationCtx.

Appendix 2: BabyIDE, the Loke integrated development environment

BabyIDE is an implementation of the Loke conceptual model as well as Loke/Expert. The DCI programming paradigm forms the foundation of the IDE, and it provides

Fig. 22 The interaction browser

browsers for each of its Data, Context, and Interaction kinds of projections. The browsers appear as overlays in a shared window. The browsers carry their own state so that work in a browser can be suspended and resumed at will. The browsers are:

- **Data:** *The Data Class browser* is for working with personal Resource classes.
- **Context:** *The Context Class browser* is for editing a Context as a class. A Context class declares the provided interface that triggers the execution of a system operation (use case). It also declares the private methods that bind Roles to objects during the execution of an operation.
- **Interaction:** *The Interaction browser* is for editing the interaction diagram with its Roles and the links between them. It also supports creating and editing the Role-Scripts that drive the collaboration of the Roles when they perform an operation.

The shared window of all BabyIDE browsers

All BabyIDE/expert browsers occupy the same area in the same window on the screen (Fig. 19):

1. This button closes the window.
2. A menu button, its commands are:
 - Remove this app from the system.
 - Change the app that is handled by this IDE.

- Add a new projection. (Not implemented)
 - Export this operation as a read-only HTML file.⁵³
3. The application is *BBa11PP*, Ellen's Smart Alarm clock. (The obscure *BBa11* name prefix is used to get around the lack of namespaces in Squeak. It confuses the code, and I have hidden it in this article whenever practicable.)

The data class browser

Any object with a globally unique identity and a RESTful⁵⁴ message interface can play a Role in a Context. The personal classes are resource classes that are specific for the current operation. They are edited in the Data projection. The panes in the Data class browser are numbered in Fig. 20:

4. The selected projection is *Data*.
5. The class list shows the personal Data classes. *BBa11MetOffice* is selected. The classes are personal resource classes specifically declared for the current operation. Other classes and services are handled elsewhere.
6. A multi-select presentation filter that shows the super-classes of the selected class.
7. A multi-select presentation filter that shows the method categories of the selected classes. By convention, the API methods form the provided interface of the resource.

⁵³ Example in <http://folk.uio.no/trygver/assets/BBa11PEllen/>.

⁵⁴ https://en.wikipedia.org/wiki/Representational_state_transfer.

Fig. 23 A screen dump that includes the BabyIDE personal programming interface

8. A list of methods in the selected method categories. The *expectedRainfall* method is selected.
9. A code pane for editing the Squeak code of the selected method. Note the difference between a BabyIDE RoleScript and a Squeak method. The first is a property of a Role, and the second is a property of an object.

The context class browser

A Context instance (aka a *Maestro*) forms the environment for the execution of RoleScripts. A Context Class Browser is used to edit the properties of the Maestro. Its panes are shown in Fig. 21:

4. The projection is *Context*.
5. A list of Contexts, *EllenAlarmCtx* is selected. A button toggles to the *Interaction* browser.
6. A multi-select presentation filter that shows the super-classes of the selected context class.
7. A multi-select presentation filter that shows the method categories of the selected classes. The *Role binding* category is selected. By convention, the *API* category

8. A list of methods in the selected method categories. There is one method for each Role in the *Role binding* category; it binds the Role programmatically to a resource object. *FORECASTER* is selected. The Role binding methods are always executed together as one atomic operation to ensure consistency.
9. A code pane for editing the selected method, the language is Squeak's default language for methods.

The interaction browser

The Interaction browser (Fig. 22) is where the programmer specifies how a BabyIDE realizes a use case as a network of collaborating objects. This projection answers three critical questions for the operation:

- *What are the objects?* A Role identifies a participating object and is a placeholder that gives the object a name at compile time. The Role is bound to the object at runtime.
- *How are they interlinked?* The programmer answers this question by linking the Roles in the interaction Diagram.

Fig. 24 Object [184] is actually playing Ellen's TIMER role, as will be seen later. (It is an instance of class Time, a subclass of ResourceObject, a subclass of ImageMorph)

- *What do they do?* The programmer answers this question by augmenting selected Roles with RoleScripts. The BabyIDE runtime system creates the illusion that these scripts augment the role-playing objects at runtime. In reality, the augmentation is virtual, and the objects remain untouched.

The panes of the Interaction Browser are:

4. The projection name is *Context*.
5. A list of contexts, one for each use case. *EllenAlarmCtx* is selected. A button toggles to the *Context Class browser*.
6. The interaction diagram where the programmer works with Roles (move, select, link, add, remove, and rename) and their link structure. The *FORECASTER* Role is selected.
7. A list of *FORECASTER* RoleScripts, *checkWeather* is selected.

properties with its objects and their relationships. The reader can also explore Loke's dynamic properties by studying program creation and execution. This appendix is a linearized, commented, and simplified projection of the Loke model. Alan Kay pointed out the difference between the two media⁵⁷:

The ability to 'read' a medium means you can access materials and tools generated by others. The ability to 'write' in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide.

The first version of BabyIDE as a conceptual model is completed:

*The Loke implementation
is a conceptual model of Loke for inspection and exploration.
It embodies the Loke model in an executable form.*

8. An editor for the selected RoleScript. (While a Squeak method accesses instance variables by name as declared in the class definition, a RoleScript accesses Roles by name as declared in the interaction diagram.)

Appendix 3: BabyIDE, the first Loke implementation

BabyIDE is an implementation of Loke as a non-intrusive extension of Squeak, a variant of Smalltalk.⁵⁵ The Squeak image containing the first version of BabyIDE together with other programs is freely available under an MIT license.⁵⁶

The implementation forms an executable, multidimensional, conceptual model of Loke that uses Squeak as its medium. A reader of the model can explore its static

BabyIDE, as an interactive development environment, is still in its infancy. I use it for demonstrating Ellen's smart alarm clock and for programming example applications such as ProkonPlan. Ellen's programming interface, BabyIDE/novice, is simpler and consists of the two Squeak windows shown in the screen dump of Fig. 23. On the right is the Resources window. It is like the desktop of a smartphone with its icons for cached resources. On the left is the BabyIDE window where Ellen composes her program.

⁵⁵ BabyIDE works under Squeak version 3.10.2. It is not easily converted to later versions.

⁵⁶ <http://dx.doi.org/10.17632/5xxgzv7fsp.1>.

⁵⁷ http://www.vpri.org/pdf/hc_user_interface.pdf.

Fig. 25 Ellen's resourceDictionaryUUID

Fig. 26 Ellen's time identifier object (The screen dump pictures have been edited to cover class name prefixes not used in this article.)

A side remark and confession:
 This first version of BabyIDE has been programed "by inspiration"
 and is neither bug-free nor easy to read.
 Being a one-man team,
 I had to suspend working with the BabyIDE program (fun)
 in order to write this article (a bore).

I'm a nonagenarian; my priorities are mandated by nature. The next step is to use the current BabyIDE to create an elegant, bug-free, and readable second BabyIDE that conforms to the DCI programming paradigm.

Loke, BabyIDE, Squeak, and Smalltalk are universes of objects and nothing but objects. For example, my Squeak universe of objects (my Squeak image) was a tangle of some 470.000 objects when I made the screen dump. They represent information of various kinds such as *message*, *string*,

collection, *stack*, *compiler*, and *service*. Every object is an instance of a class (also represented by an object).

Squeak's many programming tools such as the Inspector and Class Browser give excellent support for thinking and programming in the class abstraction. The *Squeak Reverse Engineering (SRE)* tools support a programmer and untangle

the tangle of objects by providing tools for creating snapshots of runtime object structures in the role abstraction^{58,59}:

- *SRE Execution Tracer*. `Object>>traceRM:levels:` is like `Transcript>>show:` with the addition of the oop that identifies the receiver and a dump of the stack to a specified Depth (Fig. 27). I use it to describe the execution of Ellen's smart alarm in subsection "The BabyIDE execution stack for Ellen's demo".
- *SRE Object Inspector*. A class is a partial description of its instances: Partial, because the description is fragmented between superclasses and also because the class neither discloses the state nor identity of its instances. The SRE Object Inspector shows the state and behavior of an object. The state is shown as the instantaneous value of its instance variables. The behavior is the methods found in its flattened class hierarchy. Figure 26 is an example.
- *SRE Context Browser*. The essence of object orientation is that objects collaborate to achieve a goal. The Context Browser is used to plot an instantaneous substructure of collaborating objects in an *object diagram*. As an example, Fig. 24 shows an object that is an instance of a class that is the subclass of another class, etc. A rectangle represents an object. The first text line shows the object's Squeak identifier, [oop], followed by the object's name if any. The optional second line starts with a colon followed by the name of the object's class. An arrow in the diagram shows a message path that represents an instance or computed variable. Notice the difference between the concrete SRE *as is* reverse engineering documentation and abstract models like UML diagrams.⁶⁰

BabyIDE objects

BabyIDE uses personal and shared resource objects as servers in client-server architectures where Loke is the client and where the servers are objects offering RESTful, self-explanatory interfaces.⁶¹ A standard like the *Universally_unique_identifier (UUID)*⁶² is expected to provide a unique identifier for each and every object in the world. In BabyIDE, they are accessed through instances of a UUID subclass:

```

1 [500]: UUSmalltalk>>send:withArgs: {wakerupper script wakeMe}
2 [1485]: EllenAlarmCtx >> to:send:withArgs:
3 [500]: UUSmalltalk(EllenAlarmCtxWAKERUPPER)>>wakeMe
4 [1485]: EllenAlarmCtx >> to:send:withArgs:
5 [2855]: UUSmalltalk(EllenAlarmCtxFORECASTER)>>checkWeather
6 [1485]: EllenAlarmCtx >> to:send:withArgs:
7 [3144]: UUSmalltalk(EllenAlarmCtxTIMER)>>waitTillMorning
8 [1485]: EllenAlarmCtx >> to:send:withArgs:
9 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:
10 [3943]: BlockContext >> ensure:
11 [1485]: EllenAlarmCtx >> executeInContext:
12 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:
13 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:
14 [3040]: InteractionRolePP >> startIn:
15 [236]: BlockContext >> newProcess

```

Fig. 27 The stack as printed in the transcript

Object subclass: #UUID

instanceVariableNames: 'resource'

Class comment:

An instance of this class represents a personal or shared object. Subclasses specialize the class for different access technologies; they specify how to trigger the object's operations and how to access its properties.

Subclasses of *UUID* implement RESTful message interfaces that include *apiMenuList* and *balloonText*:

Notice that instances of *UUID* are not wrappers but objects that know how to access the features of their *resource* whatever its access mechanism.

BabyIDE maintains a cache of personal and shared objects in a global dictionary: [2898]ResourceDictionaryUUID. The objectDiagram in Fig. 25 shows the structure of this Dictionary. The *Dictionary keys* are *Universally_unique_identifiers* and are visible as icons in the user's window.⁶³ The *Dictionary values* represent RESTful servers and are instances of a *UUID* subclass.

Ellen's personal objects are accessed through instances of *UUSmalltalk*, a subclass of *UUID*. The resource instance variable is the sole instance of a *ResourceObject* subclass.⁶⁴ Instances of other subclasses will identify shared objects, and their substructures may be different. In the future, vendors of shared objects will probably supply meta-information with their IoT products in a form that can be automatically converted to a subclass of *UUID*.

⁵⁸ SRE user manual: <http://folk.uio.no/trygver/themes/SRE/BabySRE.pdf>.

⁵⁹ The SRE toolset is in the Squeak image at <http://dx.doi.org/10.17632/5xxgzv7fsp.1>.

⁶⁰ <https://www.omg.org/spec/UML/2.5.1/PDF>.

⁶¹ https://en.wikipedia.org/wiki/Representational_state_transfer.

⁶² UUID: https://en.wikipedia.org/wiki/Universally_unique_identifier.

⁶³ For the purposes of this document, they are simple Squeak Symbols; #clock, #speaker, and #weather.

⁶⁴ BabyIDE has to be slightly modified if a *ResourceObject* class shall have more than one instance.

Fig. 28 An object's methods are stored in its class

BabyIDE personal objects

Figure 26 shows an *SRE Object Inspector* on Ellen's #clock resource. The bottom-left pane shows a multi-select presentation filter with the class and superclasses. Classes `UUID` and `UUSmalltalk` are selected and coalesced for the rest of the inspection. The bottom-right pane shows the method categories of the coalesced classes in a multi-select presentation filter, accessing is selected. The middle-left pane shows their accessing methods. `send:withArgs:` is selected; its code is shown in the middle-right pane. The coalesced instance variables are listed in the upper-left pane. The upper-right pane shows the value of the selected variable, here the [184]: a `Time` object.

We see from Fig. 25 that like all personal objects, class `Time` is ultimately a subclass of `ResourceObject`:

ImageMorph subclass: #ResourceObject
 instanceVariableNames: "
Class comment
There is one instance of each subclass that is uniquely identified by its resource ID, which is hardcoded in its resourceID method.

Ellen or her mentor programs her personal classes as subclasses of `ResourceObject`. For example, object [184] is an instance of class `TIME`:

```

ResourceObject subclass: #Time
  instanceVariableNames: "

" The API method category: "
Time>>delayFor: seconds
  "Wait for the given number of seconds. "
  (Delay forSeconds: seconds) wait.
Time>>waitUntil: timeString
  "Wait until the clock is 'hh:mm'. e.g.: TIMER waitUntil: '06:00'."
  | secondsDelay |
  secondsDelay := ((Time readFrom: (ReadStream on: timeString))
    subtractTime: Time now) asSeconds.
  secondsDelay < 0 ifTrue:
    [secondsDelay := secondsDelay + (24*60*60)].
  (Delay forSeconds: secondsDelay) wait.

" The accessing method category: "
Time>>apiMenuList
  | list source |
  list := OrderedCollection new.
  (self class organization listAtCategoryNamed: #API) do:
    [:methSel |
      source := self class sourceCodeAt: methSel.
      list add: {methSel asString. (source copyUpTo: Character cr) asString.}].
  ^list
Time>>balloonText
  | strm |
  strm := TextStream on: Text new.
  strm nextPutAll: 'Time' asText allBold.
  ^strm contents
Time>>defaultRoleName
  ^#TIMER "
Time>>resourceID
  ^#clock " Instead of a universal ID "
```

Fig. 29 Maestro runtime object structure

The BabyIDE execution stack for Ellen's demo

As is the case for most variants of object-oriented languages, a BabyIDE execution takes the form of a stream of messages flowing through participating objects. I used the `SRE traceRM:levels:` tool to capture a trace of the stack at the point in the execution where the `WAKERUPPER` sounds the alarm:

```
WAKERUPPER >>wakeMe
WAKERUPPER traceRM: 'traceRM' levels: 50.
WAKERUPPER soundAlarm.
```

When the execution of Ellen's demo reaches this stage, the stack is 15 deep and is printed in the Transcript, Squeak's standard output window (Fig. 27).

Smalltalk, and thus Squeak, has many advanced features such as its inherent reflection and the concrete realization of its own conceptual model. For example, I could print the

above list because the stack is an accessible linked list of stack frames. I will comment on each of the stack frames and begin at the bottom of the stack: the start of the execution.

level 15 [236] : BlockContext >> newProcess

BlockContext>>newProcess

"Answer a Process running the code in the receiver. The process is not scheduled."
<primitive: 19>

I opened Ellen's personal BabyIDE process with a World menu command. This stack frame has receiver=[236], a new `BlockContext` object.

level 14 [3040]: InteractionRolePP >> startIn:

```
InteractionRolePP >>startIn: startRole
| w context |
[
  color := diagram color.
  diagram color: Color green.
  (w := self world) ifNotNil: [w doOneCycle].
  context := self diagram model contextBrowser selectedClass new.
  context triggerInteractionFrom: self name with: startRole.
  diagram color: color.
  (w := self world) ifNotNil: [w doOneCycle].
]
forkAt: Processor userBackgroundPriority.
```

After programming it, I started Ellen's smart alarm clock with the Cue *'waitTillMorning'* menu command in the interaction diagram's TIMER symbol: [3040]:InteractionRolePP. This is a regular Squeak object and a regular Squeak message. The Ellens and Antons of this world have the illusion that the object encapsulates its methods and invokes them in response to received messages, an illusion that is sustained in the SRE ObjectInspector (Fig. 26). In reality, objects delegate to their class to compile, store, and execute methods (Fig. 28). Like all Squeak objects, object [159] has a link to its class: [2349]Met-Office. This class object has an instance variable named methodDict. This Dictionary binds selectors (messages) to CompiledMethods, Squeak methods in executable form. This is another example of that in Squeak, everything is represented by an object and that an object structure can go across different abstractions.

An important side effect of the startIn method was that it created an instance of Ellen's Context class, [1485]Maestro: EllenAlarmCtx: that performs many tasks during the execution (Fig. 27).

Object subclass: #Context
instanceVariableNames: 'roleMap'

Context subclass: #EllenAlarmCtx
instanceVariableNames: "

The class side represents the roles and their structure as a method that returns a Dictionary⁶⁵:

EllenAlarmCtx class>>roleStructure
^super roleStructure
at: #TIMER put: #(#FORECASTER);
at: #FORECASTER put: #(#WAKERUPPER);
at: #WAKERUPPER put: #();
yourself.

The code for this method is generated and compiled automatically when Ellen edits her interaction diagram. The declaration is later used by the compiler to find role names and permissible message links. For example, a TIMER roleScript can send messages to FORECASTER but not to WAKERUPPER.

[1485] Maestro: EllenAlarmCtx forms the environment for the execution of roleScripts. It has one essential instance variable, roleMap, a Dictionary that maps role names to resource objects at runtime. All roles are mapped together in an ensemble of methods to ensure consistency:

```
Context>>remap
"Map all roles to a Data object."
self resetRoleMap.
self class roleNames do:
  [:roleName | "All roles are mapped together."
    roleMap at: roleName put:
      (self perform: roleName ifNotUnderstood: [nil]) "Execute the mapping method."
  ].
self checkRoleMap
```

The mapping of each role is done in a method that is named after the role, e.g.,

EllenAlarmCtx>>WAKERUPPER
^ResourceDictionaryUUID at: #speaker ifAbsent: [nil]

The roles with their default names and methods were created automatically when Ellen moved resource icons into her Context. In BabyIDE/expert, the programmer does this work personally.

As part of its initialization, the Maestro executed Context>>remap to bind roles to objects. The result was the ephemeral object structure in Fig. 29.

Note the "secret" role #cc. Every Context is initialized with this role; BabyIDE uses it on level 7 to find the current player of a given role.

level 13 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:

Context>>triggerInteractionFrom: triggerRoleName with: selector
^self triggerInteractionFrom: triggerRoleName with: selector andArgs: {}

level 12 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:

Context>>triggerInteractionFrom: triggerRoleName with: selector andArgs: args

selector numArgs = args size ifFalse: [^self error: 'Number of arguments mismatch'].

"The Context lives on the stack during the execution of an application:"

self executeInContext: "Enter the world of roles and scripts."

```
[self remap.
 ^roleMap includesKey: triggerRoleName]
ifTrue:
  [^self to: triggerRoleName send: selector withArgs: args]
ifFalse:
  [self inform: 'Data object for role named ', triggerRoleName, ' is undefined. Interaction not started.'
   ^nil]
```

⁶⁵ The method represents the BabyIDE way of obtaining persistent objects without depending on a database.

The framed block will be executed in level 9.

The Context instance, [1485] Maestro: EllenAlarmCtx, was created on level 14. Now is the time to put it to work.

level 11 [1485]: EllenAlarmCtx >> executeInContext:

```
Context>>executeInContext: aBlock
ContextStack pushContextStack: self.
aBlock ensure: [ContextStack popContextStack].
```

Class ContextStack is a global stack of Contexts: A new Context instance is put on the stack when the execution of a context starts and is popped when it ends. The ContextStack also forwards some messages to the Context on the top of the stack, the current context or Maestro.

level 10 [3943]: BlockContext >> ensure:

```
BlockContext>>ensure: aBlock
"Evaluate a termination block after evaluating the receiver, regardless of whether the receiver's evaluation
completes."
<primitive: 198>
```

level 9 [1485]: EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:

We have digressed in levels 11 and 10 to initialize the Maestro before we execute the inner (framed) block in this

```
Context>>to: roleName send: selector withArgs: argCollection
| receiver roleClass compiledMethod |
receiver := roleMap at: roleName.
roleClass := self class roleClassForRoleName: roleName. "script repository class"
(roleClass notNil and: [(compiledMethod := roleClass compiledMethodAt: selector) notNil])
ifTrue:
  ["roleScript exists, execute it."
  ^receiver withArgs: argCollection asArray executeMethod: compiledMethod]
ifFalse:
  [(receiver isKindOf: UUID)
  ifTrue: "remote receiver, the UUID object will do the right thing"
  [^receiver send: selector withArgs: argCollection]
  ifFalse: "send regular Squeak message"
  [^receiver perform: selector withArguments: argCollection asArray]]
```

All message sends in the Role abstraction are handled by this method. The method transforms a message send into specialized messages to roles and objects, shared or personal. The code first tries to find a `CompiledMethod` for a `roleScript` and executes it if it exists. Else, the method forwards the message to a `UUID` or a regular Squeak object.

Squeak code and then compiles the resulting Squeak code in the regular way.

Ellen's code:

```
TIMER>>waitTillMorning
TIMER waitUntil: '06:00'.
FORECASTER checkWeather
```

*Context>>to: roleName send: selector withArgs: argCollection
is the key to the illusion that all messages are handled the same way independent of
the protocol used for message transmission and the nature of the receiver*

level 7 [3144]: `UUsmalltalk(EllenAlarmCtxTIMER)`
`>>waitTillMorning`

```
TIMER>>waitTillMorning
TIMER waitUntil: '06:00'.
FORECASTER checkWeather
```

is first transformed to:

```
TIMER>>waitTillMorning
(ContextStack playerForRole: #cc)
to: #TIMER
send: #waitUntil:
withArgs: {'06:00'}.
(ContextStack playerForRole: #cc)
to: #FORECASTER
send: #checkWeather
withArgs: {}
```

There is something strange here. The class of the message receiver is `EllenAlarmCtxTIMER`. Where does this class come from, and what does it do? The question needs a long answer. We saw in Fig. 28 that a Squeak method is compiled in the context of a class, is stored in the `methodDict` of that class, and is executed in the context of an instance of that class. This mechanism breaks down for roleScripts. A role is a name and not an object: There is no class, and the role is late-bound to an object at runtime. Finally, if the object is accessed through the Net, its implementation is inaccessible. Any implementation of Loke must deal with this dilemma. First, `roleScripts` can't be compiled as regular Squeak methods, so a new way has to be found. Second, a regular method is stored in the object's class. Here, there is

We know the `ContextStack` and the hidden `#cc-role` from level 11. (`ContextStack playerForRole: #cc`) is an inefficient way of finding the current context, the Maestro. After that, it's the `Context>>to:send:withArgs:` method known from level 8 that sends the messages called for in Ellen's code.

The output from the compiler is a `CompiledMethod` that is independent of the class that compiled it. A `CompiledMethod` can, therefore, be stored in any class as long as it has a unique name that makes it possible to retrieve it when needed.

*A Squeak CompiledMethod without reference to instance variables is pure behavior.
It can be stored in any class and be executed in the context of any object.*

no known class, and a new home for `roleScripts` has to be found.

The compilation of roleScripts

The `roleScript` compiler namespace includes the names of the current Role and the Roles that are visible from it. The compiler is a modified Squeak compiler that compiles `roleScripts` in two steps: It first transforms Role names to regular

The storing of compiled roleScripts

Context class names are unique within Squeak, and role names are unique within a Context. BabyIDE stores `roleScripts` (`CompiledMethods`) in hidden classes named by the unique concatenation `<context name><role name>`, e.g., `EllenAlarmCtxTIMER`. These classes are artifacts of the BabyIDE implementation. They are very special

classes with no instances, no own methods, and the personal programmer need not know about them.

The execution of roleScripts

The execution of the roleScript was triggered in the key method `Context>>to: roleName send: selector withArgs: argCollection` in level 8.

level 6 [1485]: EllenAlarmCtx>>to:send:withArgs:

see level 8

level 5 [2855]: UUsmalltalk(EllenAlarmCtxFORECASTER)>>checkWeather

```
FORECASTER>>checkWeather
FORECASTER expectedRainfall = 0
ifTrue: [WAKERUPPER wakeMe]
```

see level 7

level 4 [1485]: EllenAlarmCtx>>to:send:withArgs:

see level 8

- *level 3 [500]: UUsmalltalk(EllenAlarmCtxWAKERUPPER)>>wakeMe*

```
WAKERUPPER>>wakeMe
WAKERUPPER traceRM: 'wakerupper script wakeMe' levels: 50.
WAKERUPPER soundAlarm.
```

see level 7

- *level 2 [1485]: EllenAlarmCtx>>to:send:withArgs:*

see level 8

level 1 [500]: UUsmalltalk>>send:withArgs: {wakerupper script wakeMe}

```
WAKERUPPER >>wakeMe
WAKERUPPER traceRM: 'traceRM' levels: 50.
WAKERUPPER soundAlarm.
```

Finally, the stack was dumped on the Transcript, and the alarm sounded.

References

Web references are given in footnotes

1. Bluemke, I., Stepien, A.: Experiences with DCI Pattern. Springer, Berlin, pp. 87–96 (2015). https://doi.org/10.1007/978-3-319-18473-9_9
2. Coplien, J.O.: Lean Architecture for Agile Software Development. Wiley, Chisester (2010)
3. Dahl, O.-J., Nygaard, K.: Simula 67. Common Base Definition. Norwegian Computing Center, Oslo (1967)
4. Dijkstra, E.: The Humble Programmer (Turing Award Lecture). Communications of the ACM, October, pp. 859–866 (1972)
5. Dijkstra, E.W.: I: J. Buxton & B. Randell, red. Report on a conference sponsored by the NATO Science Committee, Rome, Italy. Rome, Italy: NATO Science Committee, p. 16 (1969)
6. Emery, F., Thorsrud, E.: Democracy at Work: The Report of the Norwegian Industrial Democracy Program (International Series on the Quality of Working Life). s.l.:s.n (1976)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994). ISBN 0-201-63361-2
8. Goldberg, A., Robson, D.: Smalltalk-80, The Language and Its Implementation. Addison-Wesley, Reading (1983). ISBN-13: 978-0201136883; ISBN-10: 0201136880
9. Hameed, S., Kahn, F.I., Hameed, B.: Understanding security requirements and challenges in internet of things (IoT): a review. J. Comput. Netw. Commun. **2019**, 9629381 (2019). <https://doi.org/10.1155/2019/9629381>
10. Hoare, C.: The Emperor's Old Clothes. *Communications of the ACM*, February, pp. 859–866 (1981)
11. Kay, A.: A Personal Computer for Children of All Ages. ACM Digital Library, Boston (1972)
12. Kay, A.: A Personal computer for children of all ages. In: ACM '72 Proceedings of the ACM Annual Conference—Volume 1, 1 August, p. Article No 1 (1972)
13. Kay, A.: The early history of smalltalk. In: ACM SIGPLAN Notices archive, March, pp. 69–95 (1993)
14. Reenskaug, T.: Administrative control in the shipyard. Tokyo, Japan, ICCAS conferences. Also at <http://folk.uio.no/trygver/1973/iccas/1973-08-ICCAS.pdf> (1973)
15. Reenskaug, T.: Prokon/plan—a modelling tool for project planning and control. IFIP Congress, pp. 717–721 (1977)
16. Reenskaug, T.: Working With Objects: The OORAM Software Engineering Method. Manning, Greenwich (1996)
17. Reenskaug, T.: Expert' voice: the BabyUML discipline of programming. Softw. Syst. Model. **5**(1), 3–12 (2006)
18. Reenskaug, T.: Expert Commentary: The Case for Readable Code. I: Computer Software Engineering Research, pp. 3–8. Nova Science Publishers, New York (2007)
19. Valdecantos, H.A.: An empirical study on code comprehension: DCI Compared to OO. Thesis, Rochester Institute of Technology (2016). <https://scholarworks.rit.edu/theses/9245>
20. Wirfs-Brock, R., McKean, A.: Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, Boston (2003)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Trygve M. H. Reenskaug is a professor emeritus of informatics at the University of Oslo. He has 60 years of experience in software engineering and R&D for industrial-strength SW products. He has extensive teaching and speaking experience including keynotes, talks, and tutorials. His firsts include end-user programming, structured design, and database-oriented architecture (1960); object-oriented applications (1973); Model-View-Controller (1979); OOram role modeling method and tool from

1983; and the premier book on role modeling in 1995. He has recently been working on DCI, a new programming paradigm that separates the code describing system state (D) from the code describing system behavior (CI). He is now working on an extension of DCI he calls Personal Programming, a programming paradigm for everybody. It adds a new abstraction layer that protects the programmer from the intricacies and limitations of the von Neumann computer below it.