Personal Programming

Trygve Reenskaug
Department of Informatics, University of Oslo;
trygver at ifi.uio.no.

Abstract

My objective is to create an intuitive computer for laypeople who want to go beyond ready-made apps to create personal programs for private needs and preferences. I submit Loke, a new personal computer that is a universe of objects that encapsulate state and behavior. The encapsulation is an inherent security barrier that separate an object's provided message interfaces from its implementation. I have implemented Loke in Squeak, a variant of Smalltalk. The implementation is an extensible and executable conceptual model of Loke for execution, inspection and exploration.

Loke was first used to demonstrate how Ellen, a novice, programs a smart alarm clock[1] through a GUI adapted to her competence, needs, and preferences. Informal demonstrations indicate that laypeople immediately grasp the idea of objects that represent concrete things in their environment and that they were creative in identifying personal opportunities for Loke and in sketching out their implementations. Interestingly, trained programmers watching the demonstration appeared to be well served with their current technology and did not see the point of Loke.

Loke qua conceptual model is completed. The model underpins its inherent security and privacy and sustains its object and message models. Loke qua programming environment is still in its infancy. Loke it is still 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 it in its own hardware could serve as both client and server in an IoT client-server architecture.

255 words

keywords:

Personal Programming, Laypeople Programming, IoT, Smart Home, MVC, DCI, BabyIDE, Smalltalk, Squeak, Object Orientation

[1] A short video illustrates the novice IDE: http://folk.uio.no/trygver/themes/Personal/TrygveReenskaug-PersonalProgramming(AVCHD.H264.1440x1080p24).mp4

Table of Contents

Preface to the draft program and article

It all started with a hunch, an idea, an intuition, or whatever turns up in my mind. The next step was to express it formally. I use Squeak, a dynamic, multi-dimensional medium, to model my intuition. The model is expressed as an executable structure of objects. There is no source code in closed from as is usual for conventional programing languages, the program *is* the object model: It can be explored by inspection and execution in different projections that highlights interesting aspects of the model.

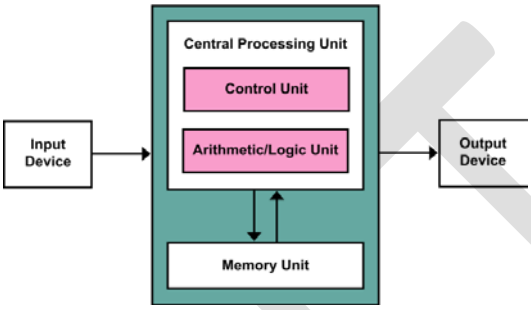
The third stage is the hardest and takes most of the effort. The task is to transform the rich Squeak multi-dimensional model into a unidimensional document; a daunting and almost impossible task. I now post a snapshot of the current state of the Squeak model and draft document and open the process for comments. The model is posted as a Squeak image and a document in http://folk.uio.no/trygver/themes/Personal/pp-index.html. Work is continuing to make the document ready for publication, hopefully in SoSyM.

Oslo, June 2019.
Trygve Reenskaug

A. Prologue

Modern computer programming celebrated its platinum jubilee on the 21st of June, 2018. Exactly 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 the computer, affectionately known as *Baby* , followed the then well-known von Neumann model (Figure 1). *Baby* is morphing into billions of communicating computers, many of them connected into *a single, global machine.* (Kelly, 2008).

Figure 1: The von Neumann model forms a solitary system.
~~vonNeumann-3.png~~



Like a computer, *an object* is an entity that encapsulates state and behavior and has a globally unique identity. Object state is in the form of references to other objects. Object behavior is triggered by received messages. The object does not know why a message was sent, i.e. its context. Object behavior is triggered by received messages. The triggered behavior can cause a change in the state of the object and also make the object send messages to other objects without knowing what their receivers will do with them. In short, the object is stand-alone and unaware of its context.

In software engineering and computer science, abstraction is the process of removing physical, spatial, or temporal details or attributes in the study of objects or systems in order to more closely attend to other details of interest[2]. A common abstraction on objects in programming and computer science is the *class abstraction.* It is supported by a multitude of languages such as Java and Ruby:

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

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:

1. An object is an entity with an immutable and globally unambiguous identifier[3].
2. The object provides an interface that can be invoked by messages from other objects.
3. The object plays a role in its collaboration with other objects.
4. The object's encapsulation hides its inner construction.

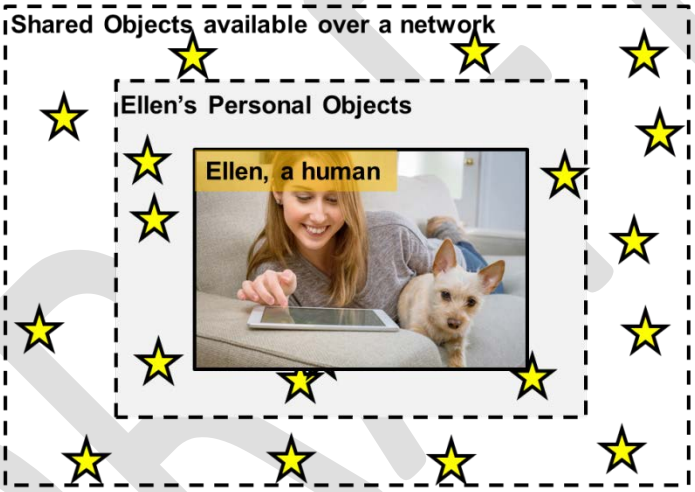
[2] https://en.wikipedia.org/wiki/Abstraction_(computer_science)
[3] See for example https://en.wikipedia.org/wiki/Object_identifier

The class and role abstractions are complementary: what's exposed in in the class abstraction is hidden in the role abstraction. Conversely, what's exposed in the role abstraction is hidden in the class abstraction.

My style of research is to experiment with software in my laboratory. I considered two alternative foundations for this laboratory. The first was a von Neumann machine. I rejected this model because its model lacks the essential peer-to-peer communication of modern computing. I landed on *Squeak*, a variant of Smalltalk, because it is a universe of communicating objects much like the single, global machine.

I merge the objects of a distributed machine such as the Internet with a personal universe of objects such as the objects of my Squeak laboratory. I get the *personal object computer* of Figure 2. I name it Loke after a powerful and somewhat deceitful god in Norse mythology who sometimes assists the gods and sometimes works against them.[4] The Loke computer is personal and private in the sense that a smartphone is personal and private. It is an instance of a general platform that is personalized by augmenting it with data such as address books, bookmarks, letters, and personal programs according to the needs and preferences of its owner.

Figure 2: Ellen's Loke with her personal and public objects.
~~ObjectComputer-4.png~~



The ground rules for Loke are[5]:

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

The instruction repertoire of the *Baby* computer was realized in electronic circuits that interpreted its instructions. Today, a typical von Neumann computer has replaced some of the hardware circuits with *microprograms* that realize the computer's instruction repertoire. Similarly in Squeak, the instruction repertoire of an object is realized by methods declared in its class:

Squeak classes can be seen as the microcode of object-orientation.[6]

[4] Loke is sometimes called *Loki*. He can take on many different shapes according to whim. https://en.wikipedia.org/wiki/Loki#Scandinavian_folklore
[5] Inspired by (Kay, 1993)

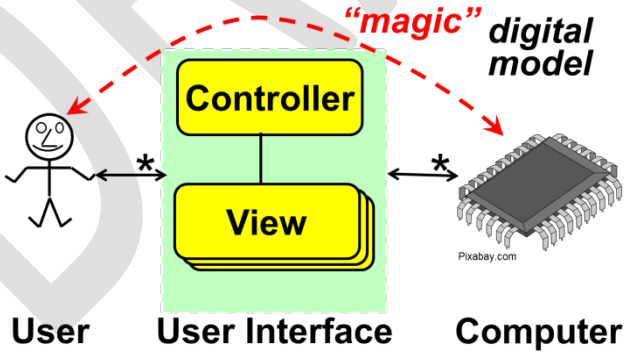
In a von Neumann computer, *a program is a collection of instructions for performing a specific task.* In Loke, the behavior of a program emerges from each of its participating objects behaving according to its nature. There is no notion of a program in closed form that controls the message flow.

The *Data, Context, Interaction (DCI) programming paradigm* embraces the notion of programming Loke: System state is declared as *Data* in the class abstraction; System behavior is achieved in the role abstraction by *Contexts* that muster participating objects to play their roles in *Interactions* [7]. This DCI model of computing is on an abstraction level above the von Neumann model.

The operators of the Manchester Baby computer worked it through a control panel. This panel has morphed into *graphical user interfaces (GUIs)* that empower each and every one of us to augment our intellect.[8] The consequences are deeply radical for individuals and society alike. We still find archaic control panels in everyday products. Two examples from my home spring to mind: My dishwasher and my electric cooker both confuse me with obscure icons and button-presses. Even worse is my local "Internet of Things" that connects my VCR and TV with their multi-button remote controls.

Figure 3 illustrates Model - View - Controller (MVC), a conceptual model of a GUI. (Reenskaug, 1978). The User experiences MVC as an extension of their mind. The "magic" of Figure 3 is achieved by first making the computer's digital model correspond to the user's mental model of the domain and second by choosing well-known representations for presenting the Model information in the Views so that it can be readily intuited by the *User*. An invisible *Controller* 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 computer fades into the background in the user's mind.

Figure 3: MVC
~~MVC4.png~~



A program for the *Baby* computer was a sequence of instruction that were copied into its memory, and the execution started with its first instruction. The general workflow has essentially remained unchanged from the beginning to this date: *Write the program text; load it; and execute it on a von Neumann computer*. Modern GUIs have changed the nature of computer use for almost all categories of users. A notable exception is a large group of mainstream computer programmers who use text only and have excluded tables, graph-based representations, etc., as viable media for program specification.

---

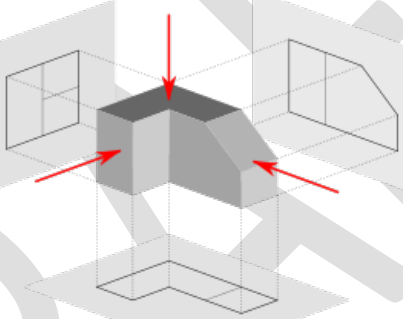
[6] https://en.wikipedia.org/wiki/Microcode
[7] https://en.wikipedia.org/wiki/Data,_context_and_interaction
[8] https://en.wikipedia.org/wiki/Graphical_user_interface#History

In this article, I explore how the DCI model of computing can be combined with MVC-based programming tools to help a broader category of computer users master computer programming. I call them *personal programmers* to distinguish them from the professionals. While a Professional Programmer is a highly trained expert, the Personal Programmer writes programs for personal use on their Loke and values simplicity and convenience over programming sophistication. I propose that they are best served by a multi-media Integrated Development Environment such as BabyIDE. The objects of the resulting programs are distributed among participating objects and are represented as an MVC Model.

Some of the Model's objects will be personal, some will be shared; some will be immutable, some will be mutable. Instances of Squeak library classes are assumed immutable since the set of objects involved in an execution is unknown. It is a research project to minimize the threat the integrity of Loke caused by new Squeak releases.

Views on the Model let the programmer work with the program seen in different projections. The projections are like an engineering projections with their plan and side views of a solid (Figure 4).

Figure 4: Engineering drawing of a solid in three projections.[9]
~~First_angle_projection.png~~



Different parts of the Model can be handled by different Views, either pre-defined or created automatically. Variants of BabyIDE can be created for different categories of users. Novice users can enjoy a low entry threshold with simple Views while an expert user will appreciate the power of expression of a high abstraction level. Both build on the same conceptual Model and the transition from novice to expert can be a smooth growth guided by the Personal Programmer's increasing experience and needs.

This article is an interim report from an ongoing project. The rest of the article is organized in four parts and two appendixes:

Part B: Novice Programming. Ellen, a novice Programmer, has a simple problem that she solves through a simple BabyIDE View on the program. I illustrate how she programs a smart alarm clock that will wake her in the morning if the weather forecast promises a dry day (0). An amateur video illustrates the idea. Ellen's DCI-based mental model of computing will evolve in her mind with programming experience (B.2). Her smart alarm clock heralds a new way of programming a structure of communicating objects rather than a stand-alone CPU as we find it in a von Neumann machine (B.3).

Part C: Loke. I discuss the nature of Loke, its model, and how it how it can be used.

[9] Picture ©Emok 2008, https://commons.wikimedia.org/w/index.php?curid=4280545 reproduced by permission

Part D: BabyIDE, the Loke implementation. BabyIDE is implemented as a non-intrusive extension of my Squeak laboratory. The DCI software comes at the top of an abstraction hierarchy starting with the hardware microcode and ending with the DCI Interaction.

Part E.1: Further work. Some important tasks that remains to be done.

Part E: Epilogue

.

Part E.2: Related work. Other initiatives that are related to personal programming,

Part E.4. Acknowledgements

Part E.5: References. Web references are in footnotes.

Appendix 1: BabyIDE User Manual. A detailed description of BabyIDE with its tools

Appendix 2: ProkonPlan, an Example. A comprehensive BabyIDE programming example with an architecture that combines MVC with DCI.

B. 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 immediate 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 better understand a physical phenomenon. 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. All of them want to take control over their environment with its devices and services; they need to become computer literate.

I do not know how many personal programmers there will be in the world, but I choose to aim the design for some tens of millions of them. This makes it impracticable to give each of them formal training in a 3GL. I replace the von Neumann computer with *Loke*, a new and more intuitive alternative. I single out Ellen to represent all novices and give her *Loke*, a *personal object computer*, is on an abstraction level above van 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.

People experience their computer applications through their user interfaces: The user interface *is* the program. I have experimented with an IDE for the Ellens of this world and call it *BabyIDE/Novice*. I have three requirements:

1. *BabyIDE/Novice* shall provide a low entry threshold by building on an idea that is well known to the novice programmer. The IDE shall protect the programmer from the complexities of the hardware and its programming languages by working on a higher abstraction level. I have chosen the idea of *composition* for its foundation; an idea I believe is shared by most if not all humans. (The well-known Lego toy lets children compose their projects from small bricks).
2. *BabyIDE/Novice* shall help the novice build a comprehensive mental model of computing. This is achieved by displaying the generated code.
3. Problems to be solved with *BabyIDE/Novice* shall scale from the first, small projects of the novice to the larger projects that they can tackle as they learn to become an expert. This is achieved by building on the Loke model of computing.

I have created a proof-of-concept implementation of BabyIDE/Novice to demonstrate its principles. This Part describes a demonstration of the program using the terminology of the novice but aims its contents at the mentor who will demonstrate it. Readers of this part are expected to have intuited the idea of an algorithm from using the Scratch programming language or similar[10].

[10] https://en.wikipedia.org/wiki/Scratch_(programming_language)

B.1. Ellen's Smart Alarm Clock

Households are in danger of being overwhelmed by disruptive smart home technologies (Figure 5). I expect there will be many such households. I design my solution for millions of home owners and single out Ellen to represent any novice programmer. Ellen owns a smart home with an Internet of Things (IoT) that connects to her many Things. The IoT is again connected to the Web and thence to the whole Internet. This gives Ellen access to a vast number of resources that she can muster to serve her needs.

Figure 5: Ellen's Internet of Things
~~PP-ProgrammingForAll.png~~

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. So, she needs to create an alarm clock that checks the weather forecast before it wakes her, but first she needs to learn how to program it. She needs a solution that is easily learned: I can't expect millions of people to attend extended courses. The Danish philosopher Søren Kierkegaard has given this excellent advice:

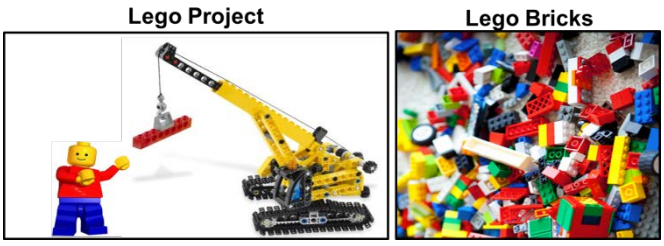
> If One Is Truly to Succeed in Leading a Person to a Specific Place,
> One Must First and Foremost Take Care to Find Him Where He is and Begin There.[11]

So the critical question is: "Where can I find the Ellens of this world?". I tried to find a common ground for all humanity and looked up psychological research into the behavior of young babies. I didn't find any good answers. There was no consensus and the research 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 is a good candidate. Most children have probably created a cow by sticking four pins into a cork. The composition of LEGO® bricks into what Lego calls *projects* is an experience shared by many (Figure 6):

> Lego products are sold in 130 countries. UN has 193 Member states. 75 billion LEGO bricks are sold per year, about 10 for each of the world population of 7,6 billion. On the average, each person on Earth owns 75 Lego bricks.[12]

[11] https://teol.ku.dk/skc/sab/citater/
[12] https://www.telegraph.co.uk/finance/newsbysector/retailandconsumer/8360246/Lego-in-numbers.html

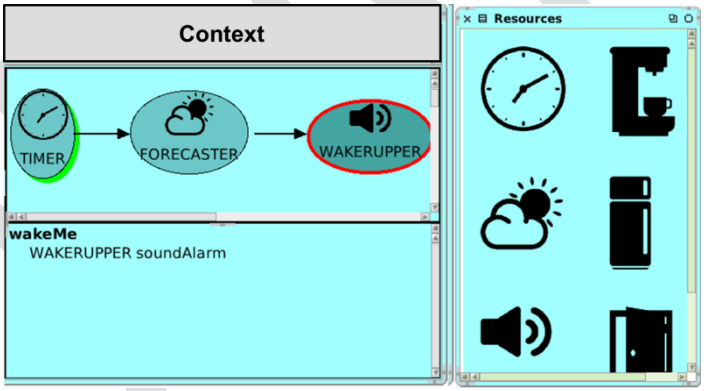
Figure 6: A Lego construction toy
~~OO-Lego-3.png~~



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

Ellen's box of objects appear as icons in the window called "Resources" in Ellen's IDE. (Figure 7, right). She selects one by one and drags it into her workspace in the window called "Context" in the IDE (Figure 7, left). She names it according to the role it plays in her project and creates a road map by connecting it to other objects. This is a client-server connection as is assumed in most IoT communication standards. It is important 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.

Figure 7: Ellen's IDE.
~~PP-Ellen-DemoTool-8.png~~
NOTE Show clock balloon= Timer or Timer?



Ellen first selected an object that represents the loudspeaker that has replaced her old alarm clock. She picks it up, moves into her project, and names it WAKERUPPER. 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 object with a script that tells it what she wants it to do for her:

```
wakeMe
    WAKERUPPER soundAlarm.[13]
```

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:

```
checkWeather
    FORECASTER expectedRainfall = 0
        ifTrue: [wakerupper wakeMe][14]
```

[13] This is Smalltalk syntax, chosen for being more accessible to the novice.
The Java equivalent is
void **wakeMe()** {WAKERUPPER.soundAlarm()}

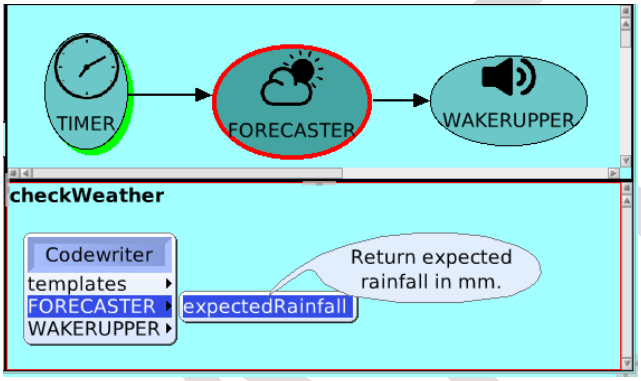
Finally, she selects a timer that will perform this program in the morning:

```
waitTillMorning
    TIMER waitUntil: '06:00'.
    FORECASTER checkWeather.[15]
```

That's it, the program is complete and Ellen sets her alarm clock by activating this first roleScript.[16]

I have chosen to give Ellen a menu driven input interface so that it only requires her to be able to read the code; making it part of her passive vocabulary. This preliminary solution is in the form of a two-level menu of code templates (Figure 8). A balloon text helps Ellen when the mouse hovers over a menu item.

Figure 8: Ellen's Mental Model of computing
~~PP-InsertMenu.png~~



A benefit of this form of visual programming is that the generated code is visible and editable, thus helping Ellen to gradually include textual programming of roleScripts in her active vocabulary.

B.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, taken together during an execution, constitutes Ellen's program. Her three *Views* (Figure 9) 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.

---

[14] The Java equivalent is

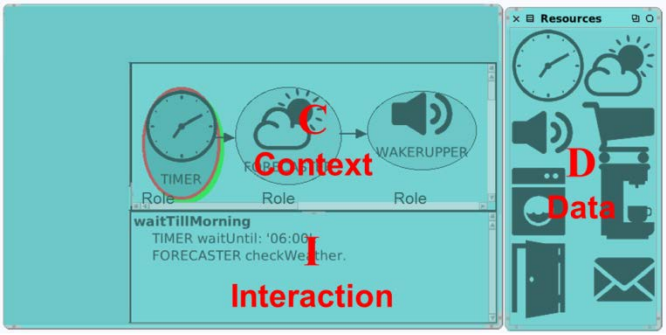
```
void checkWeather() {
    if (FORECASTER.expectedRainfall = 0) {
        WAKERUPPER.wakeMe;
    }
}
```

[15] The Java equivalent is

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

[16] You can find a video of the process here.

Figure 9: Ellen's DCI window for Personal Programming.
~~PP-window-timer-1.png~~



DCI gives Ellen a powerful separation of concerns in three orthogonal projections. They are called *Data*, *Context*, and *Interaction*:

- The **Data** are stand-alone objects such as the objects that represent things in Ellen's home.
- A **Context** is a structure of selected objects that play their roles to meet Ellen's goal (use case).
- An **Interaction** is the code within a Context that augments the behavior of the participating object to make things happen.

There is also a projection of the program that presents it as a text. The text for Ellen's smart alarm clock is at http://folk.uio.no/trygver/assets/BBa11PPEllen/readableVersion.html.

A theatre 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 theatre'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*" (Figure 10). In DCI, the roll of paper is called a roleScript.[17]

Actors play their roles together with other actors on a stage; data 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.

Figure 10: actors and Interaction roles perform their roleScripts.
PP-roleScript.png

[17] https://en.oxforddictionaries.com/definition/us/role

B.2.1. Ellen's Data

In the Introduction, we argued for basing Ellen's mental model of computing on the Loke computer (section C). Its most important 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 (Figure 9, right) are objects of her Loke computer; 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:

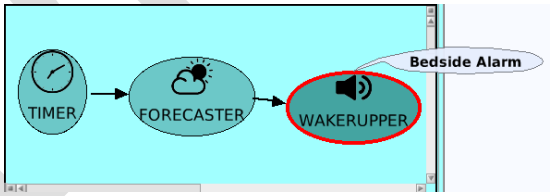
- What are the objects?
- How are they interconnected?
- What do they do?

The questions are open-ended and many professional programmers find it hard to answer them. (Wirfs-Brock & McKean, 2003) discuss various techniques that help them find good answers. It is easier for Ellen because all her objects are pre-defined and her program is for fulfilling her immediate needs here and now. The point is that she perceived a need, poked around in her catalogue of existing objects to select the ones that could help her, and then composed a program that served her needs. This is much more concrete than if she started from scratch and had to invent the objects she needed. Ellen is only interested in what the objects can do for her; i.e., their provided interfaces. The way they are implemented, for example with classes, is irrelevant to her.

B.2.2. Ellen's Context

Most personal programs involve more than one object that perform their roles within the DCI Context. In the context diagram (Figure 11), an ellipse represents a role. An arrow represents a link, i.e. a communication path for passing messages.

Figure 11: EllenAlarmContext
~~EllenAlarm-diagram.GIF~~



In a theatre, 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 one of her roles as long as it has the capabilities needed by the 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, handle 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, BabyIDE/Novice, remembers Ellen's gestures by generating code for casting objects to her roles:

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

As a novice, Ellen will not see this code. Expert Loke programmers, however, may want to replace it with more sophisticated methods for mapping objects to roles.

B.3. A new way of programming

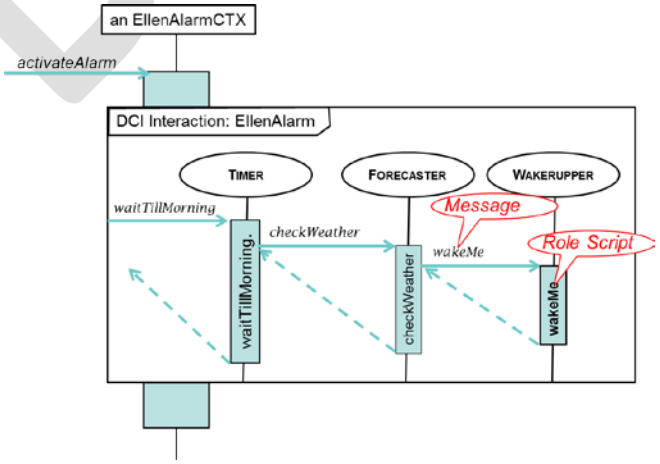
The programming tool that I have shown here is *BabyIDE/Novice*, an experimental proof-of-concept implementation. I have shown the smart clock demonstration to a farmer (my daughter) who had been using computers for many years but who had never written a program. We were brainstorming a number of possible 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 was more like setting up a playlist in a music server than programming in the conventional sense of the word. There were no self-contained source file, no discernible compilation stage, and no concrete program. Instead, Ellen selected objects from her environment and applied the DCI paradigm to control the flow of messages that met her needs. 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 Part of this article.

B.3.1. Ellen's Interaction

Ellen first selected the objects that let her compose her smart alarm clock in a Context while she envisaged the flow of messages that would realize the behavior of her clock. Her next task was to fill in the roleScripts that made this happen. (Figure 12).

Figure 12: Message sequence chart
for Ellen's *activateAlarm* system operation.
~~Runtime-MSC-2.png~~



The idea of roleScripts is new to Ellen and we endeavor to introduce it as smoothly as possible. We have chosen the Squeak default language for scripting because it is

conceptually and syntactically simple and is easy to read for the uninitiated.[18] It is hard to read for most experienced programmers because it is different from what they are used to. The main stumbling block is the syntax for messaging. A typical mainstream syntax for a procedure call is:

```
receiver.selector (arg1, arg2, ...)
e.g., fileDirectory.copy (london, paris);
```

The Squeak syntax for a message send is more informative:

```
receiver name1: arg1 name2: arg2.
e.g., fileDirectory copyFrom: paris to: london.
```

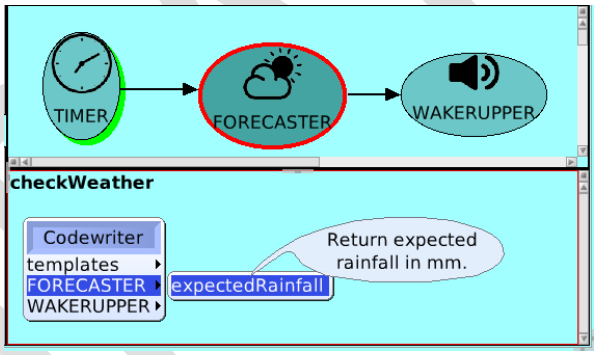
I give priority to the uninitiated and hope experts reading this will have the patience to decode our simple examples. More about the Squeak language can be found in the Squeak home page[19].

Ellen creates and edits a roleScript in a text editor (left-bottom pane in Figure 7). This is a conventional *"remember and type"* kind of user interface; it requires that the language is part of Ellen's active vocabulary. For example, she must know the language to be able to type:

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

Figure 13: Coding by menu selection

~~PP-InsertMenu.png~~



A benefit of this form of visual programming is that its generated code is visible and editable, thus helping Ellen to gradually include textual syntax in her active vocabulary.

BabyIDE is the programming interface for Loke. It always caches Ellen's personal objects as well as some shared ones. She adds more objects from Loke's shared objects as needed. This extends the expressiveness of her language without changing her fundamental model of computing or even her IDE. We 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.

[18] You are free to define a different language for your BabyIDE roleScripts if you know how to create a compiler for it.

[19] http://wiki.squeak.org/squeak/1859

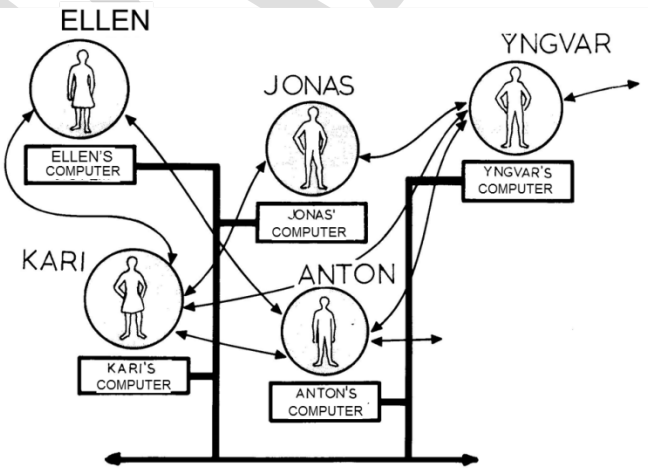
C. Loke: The Personal Object Computer

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 [20]. 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 it 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 later will succumb under the increasing weight of its own complexity.

C.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 (Emery & Thorsrud, 1976), (Reenskaug, 1973). 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 for supporting *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 communication with other managers (Figure 14). (Today, each manager will be assisted by their personal Loke). 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 Anton's computer receives a request or report, he can accept or reject it automatically or he can let it wait for a personal intervention.

Figure 14: Prokon's distributed personal computing.[21].
~~1973-08-ICCAS-2.png~~

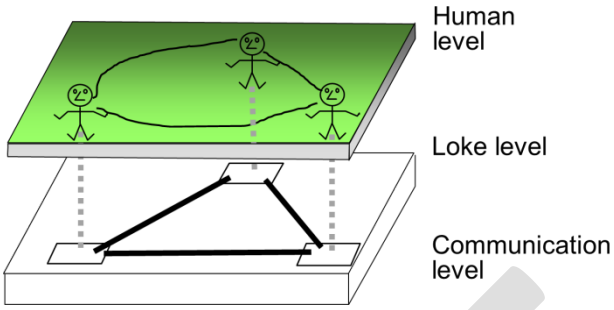


One of Anton's tasks is to create a plan for his department and to coordinate it with the plans of his colleagues. Different departments have different responsibilities: One can be responsible for designing a part, another can be responsible for manufacturing it. Both understand the logic of their different programs and ideally write them themselves.

[20] https://en.wikipedia.org/wiki/Internet_of_things
[21] In Figure 14, thick, straight lines denote computer communication and thin lines denote human communication. The figure is copied from (Reenskaug, 1973).

From time to time, managers meet to synchronize their individual plans. Their personal computers (Lokes) are running below the table in Figure 15 and communicate to support their owners who are negotiating personally above the table.

Figure 15: Humans coordinate their plans supported by their communicating Lokes
~~DistributedComputers-5.png~~



Concurrently with the Prokon project, the developments at Xerox PARC in the 70s were on a different scale in creativeness, impact, and funding [22]. One of the prime movers in this development was Alan Kay with his Dynabook ideas (Kay, 1972) and his definition of object-orientation that he verbalized much later:

> In computer terms, Smalltalk is a recursion on the notion of 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 postponed 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. (Kay, 1993)

I start from Prokon's open structure of people and collaborations and augment it with Kay's object orientation. The objects include the Squeak universe of objects and the objects found on the Net. I also supersede the implied von Neumann computer with an object computer and make it personal like a smartphone. The result is Loke. the personal object computer.

Squeak, a variant of Smalltalk, is a universe of objects that 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 independent existence and ownership even when they are not connected to the Net. In contrast, a Squeak universe of objects is closed in the sense that its objects have no existence outside that universe.

Conceptually, Smalltalk/Squeak started its execution some time in the distant past and my instance is executing a fork of that execution. The execution is paused when I store 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 is started when that copy is loaded. That means that Anton

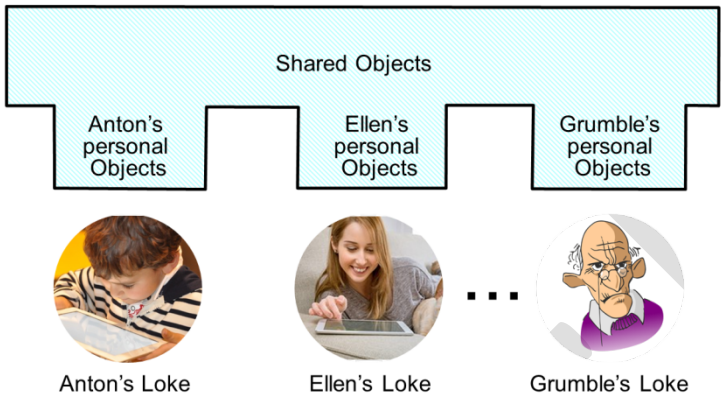
[22] https://www.techspot.com/guides/477-xerox-parc-tech-contributions/

shall not only have his personal copy of the programs but also the exclusive ownership of his personal objects and continuous execution.

C.2. The Loke conceptual model and machine

Loke was introduced in section (A: Prologue) 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. The merged objects of Loke were illustrated in Figure 2, Figure 16 gives a more detailed picture.

Figure 16: A world of Lokes
~~ObjectComputerLoke-2.png~~



I open for tens of millions of novice and expert owners of their personal instances of Loke. It is not realistic to expect that many people receiving formal training in programming, so I have created Loke to be as simple and intuitive as I possibly can without loss of expressive power.

- 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 provided message interface.
- An object is either shared or personal.
- The objects have different representations, owners, 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

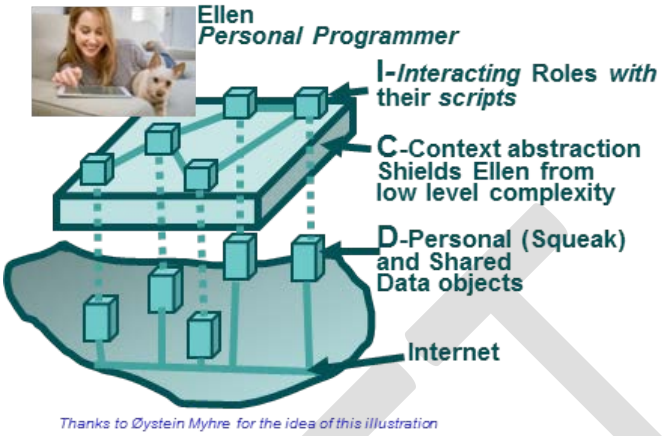
Ellen refines her understanding of her Loke by programming it. Just as a smartphone user accumulates apps, Ellen accumulates a fund of personal and shared objects that cover her various domains of interest. It can be said that while a domain specific language specializes a computer for a certain domain, Ellen's use of her Loke specializes it for her own needs, things, and preferences.

C.3. DCI: The Loke programming paradigm

When Ellen programmed her smart alarm clock in section B.1, she started the clock with a menu commmmand 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 methods and objects to DCI with its context, roles, and roleScripts. Figure 17 illustrates how the Context with its roles and scripts form a new level of abstraction above Squeak. A role can only send messages to itself, the roles it is linked to, and the object it represents. Objects do not know the roles that play them. This restricted

visibility is an important feature of the DCI strong separation of concerns that shields the Loke programmer from the idiosyncrasies of Squeak and the networked programming levels below it. The argument also works in the opposite direction: The Context abstraction protects the lower system layers from possibly malicious attacks from Ellen and her friends.

Figure 17: The communicating roles in Loke are on a new abstraction level.
~~DCI-abstractionLayer-4.png~~



C.4. Reliability, security, privacy

There is an increasing concern about the reliability, security, and privacy for information systems in general and IoTs in particular. (Hameed, et al., 2019). My own understanding of the multitude of challenges and solutions is almost non-existent, but I am able to observe my own PC: My hardware is a von Neumann machine with its central processing unit (CPU) and memory (Figure 1). A program resides in the memory and the CPU executes whatever it finds there so my PC is basically 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 XXX offered a very interesting program for free trial. During the downloading process, my OS asked: "*Do you permit XX to access to your system?*" I had to answer YES while wondering why XXX must be allowed to secretly plant any malware into my system and why a less comprehensive permission couldn't suffice. What I have is an basically 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 all the fascinating software offerings on the Web, of course).

Loke goes to the root of the problem by superseding the van Neumann computer by Loke, the safer Object Computer alternative. Loke is designed to operate on an abstraction level immediately above the hardware and the OS is replaced by ensembles of collaborating objects. I defined the nature of the Loke object in section A: "*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 inherently separates the outside of an object from its inside. The outside is a possibly evil world that can only access the object through its provided message interface. The inside handles 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 ends when an incoming message is handled in some other way. An example: Layers of protection can prevent an IoT Thing from worming its way into the heart of a corporate information system.

I have read somewhere that all developers of IoT-related programs should understand 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 problems that should have been resolved by the Loke implementer and the IoT communication providers. Nevertheless, Loke must be able to handle exceptions that can arise from below without confusing its owner. A common 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.

A final word: I do not believe that Loke solves all reliability, security, and privacy challenges, far from it: The majority will remain. What I hope is that Loke removes a few and helps thinking about the rest.

C.5. Programming by thinking

A pre-requisite 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 and my programs were a few lines of binary code. Yet they 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. I couldn't get a better mind and had to look at my programming habits. I first considered *"programming by testing"* and wrote short loop within a loop on the back of an envelope. Complete testing of all possible executions would take a few hundred years, so I quickly abandoned this approach. In practice, testing can never be complete and Edsger Dijkstra states the obvious in this famous quote from (Buxton & B. Randell, 1969) p16:

> Testing shows the presence, not the absence of bugs.

I ended up with *"programming by thinking": Any method that prevents a programmer from writing code, is a good method*. This led to a simple architecture with a clear separation of concerns with a central database surrounded by application programs. Each application was decomposed into a subroutine call tree. I could now wrap my mind around the whole and each of the parts separately. On this foundation, my team built a 50.000 lines program for the computer-aided design and manufacturing of ships that proved to show relatively few bugs when was installed in most of the world's major shipyards. The key was the simplicity of its architecture and subroutines.

Much later, Loke is programmed according to the DCI paradigm, another method for "programming by thinking". The Data are stand-alone objects that can be developed independently. The Contexts with their roles, 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 making code more comprehensible and thus appear simpler than comparable Java code. (Valdecantos, 2016). Other examples of using DCI to improve simplicity are (Bluemke Ilona, 2015) and a DCI-structured program for managing book loans in a library [23].

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

[23] A library program: https://github.com/ciscoheat/haxedci-example

,“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, 1981)

C.6. Get it right the first time

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 code writers and reviewers learn from the experience [24]. At the time of the Byte article, my team was creating a medium-sized 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 database schema and the program call tree. The unit for review was the subroutine: Each subroutine was assigned 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 were it was assigned to another team member for review. It was tolerated and even expected that people make mistakes when coding. It was 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. The remaining subroutines had only minor bugs. No deficiencies were found during system testing or in the program's lifetime. 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 manpower for such elaborate development processes, but they do find time for lengthy testing and rework.

This was our only experiment and the program was very small. The experience was promising and the team was motivated to try it out on a "real" example. Unfortunately, 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 to the classes of collaborating instances. We could not find chunks of code that could be independently specified, written, reviewed, documented, and tested.

Software engineering is a very special branch of engineering. Normal documentation is annotated with two signatures: *Created by (date and initials)* and *Checked by (date and initials)*. Object oriented programs can have the first signature but they can't be read and checked. To quote the *Design Patterns* book (Gamma, et al., 1994) 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 common way of programming.

Today, some 50 years after my first experience, code review is again feasible and it would be interesting to try it out on 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 stand-alone objects. The orthogonal role abstraction gives independent chunks of

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

code for the definition of system behavior. Peer review is now feasible. Dijkstra put it this way (Dijkstra, 1972):

> 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.

D. BabyIDE, the Loke implementation

Readers of this section like studying code
and are acquainted with the Smalltalk/Squeak notation and execution model.
Everyone else may skip it.

ST/Squeak defined in itself -> inherent reflection.
sometimes hard for my brain to navigate along object structures in 3 independent dimensions:

- build time objects - run time objects
- class objects subclass of superclass objects etc.
- object instance of class object which is instance of etc.

BabyIDE is an implementation of Loke as a non-intrusive extension of Squeak[25], a variant of Smalltalk. 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 properties with its objects and their relationships. (D.1) The reader can also explore Loke's dynamic properties by studying program creation and execution.(D.3) This article is a linearized, commented, and simplified projection of the Loke model. Alan Kay pointed out the difference between the two media:[26]

> 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.

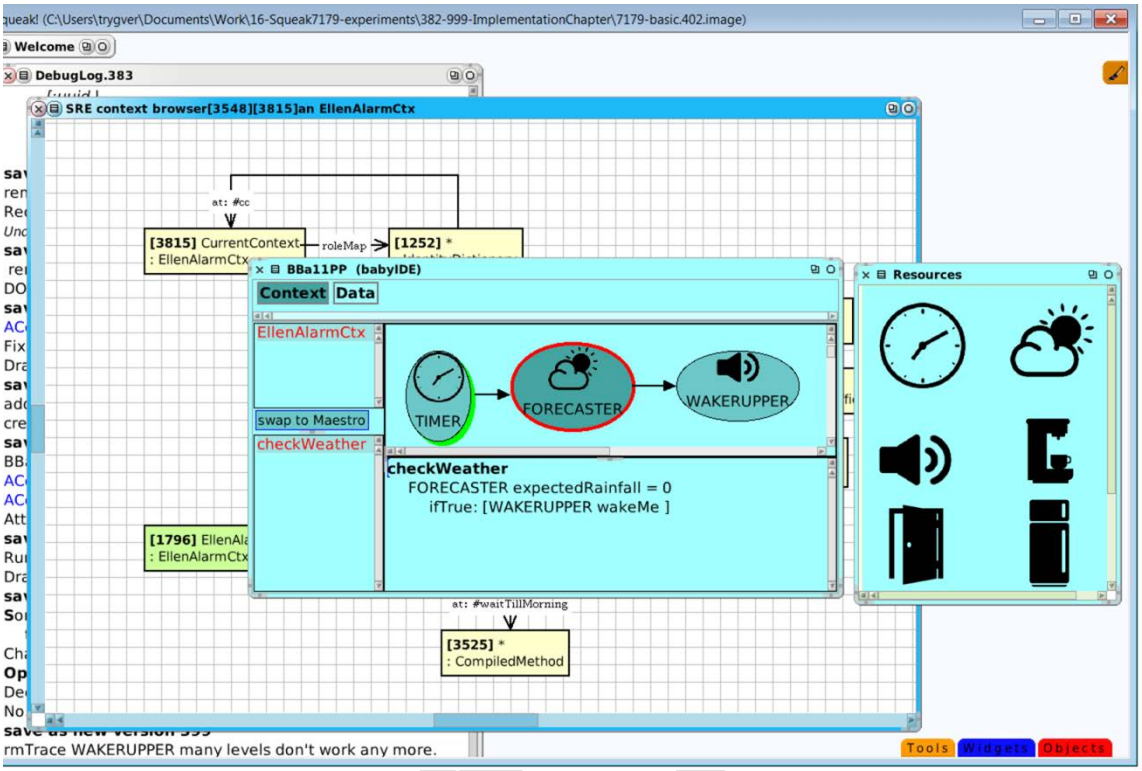
A first version of BabyIDE as a conceptual model is now completed. It will never be finalized because BabyIDE is my laboratory for improving it.

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

BabyIDE as a personal computer with an interactive development environment is still in its infancy. I use it for the demonstrating Ellen's smart alarm clock and for exploring other applications. Ellen's programming interface consist of two Squeak windows as shown in the Squeak screen dump of Figure 18. On the right is the Resources window. It is like the desktop of a smartphone with its icons for the cached resources. On the left is the BabyIDE window where Ellen composes her program. A detailed description of the BabyIDE development environment is in Appendix 1: BabyIDE User Manual.

[25] BabyIDE works under Squeak version 3.10.2. It is not easily converted to later, more complicated versions.
[26] http://www.vpri.org/pdf/hc_user_interface.pdf

Figure 18: My Squeak environment
with the BabyIDE personal programming interface.
~~LokeEmbeddedSqueak-4.png~~



Loke, BabyIDE, Squeak, and Smalltalk are universes of objects and nothing but objects. For example, my Squeak universe of objects was a tangle of some 470.000 objects when I made the screen dump. They can be characterized by abstract terms such as *message*, *string*, *collection*, *stack*, *compiler*, *service*, etc. etc. Every object was an instance of a class (alo 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 untangle the tangle in the role abstraction by providing tools for working with runtime structures of collaborating objects:[27]

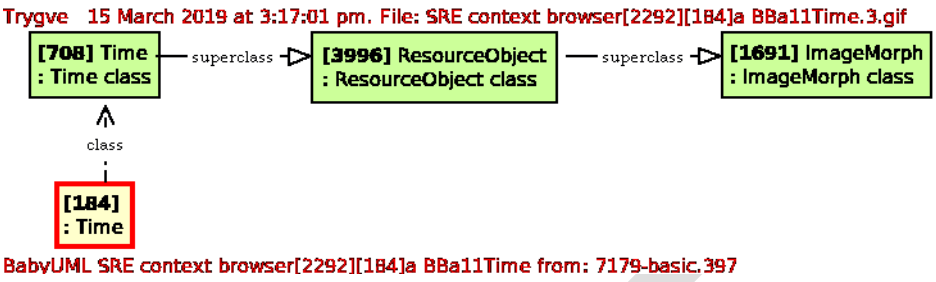
- *SRE Execution Tracer*. Object>>traceRM:levels: is like Transcript>>show: with the addition of the oop that identifies the writing object and a dump of the stack to a specified depth. I use it to describe the execution of Ellen's smart alarm in section D.3.
- *SRE Object Inspector*. A class is a partial description of its instances. Partial, because the description is fragmented between superclasses and also because the class does not disclose the state or 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 21 in section D.2 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, Figure 19 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 second line

[27] SRE user manual: http://folk.uio.no/trygver/themes/SRE/BabySRE.pdf

starts with a colon followed by the name of the object's class. An arrow in the diagram shows a message link that represents an instance or computed variable. Notice the difference between the concrete SRE *as is* reverse engineering documentation and the more usual abstract models like UML diagrams[28].

Figure 19: Class Resource [184] is a kind of Time: A kindOf ImageMorph

~~SRE Context browser[2292][184]a BBa11Time.3.gif~~



D.1.Loke objects

BabyIDE uses Loke objects as servers in client-server architectures where BabyIDE is the client and where the servers are objects offering RESTful, self-explanatory interfaces.[29] Ellen utilized such an interface when she programmed her roleScripts in Figure 13. A standard like the *Universally_unique_identifier (UUID)*[30] 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:

```
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*: Messages that were used to create Ellen's coding interface in Figure 8.

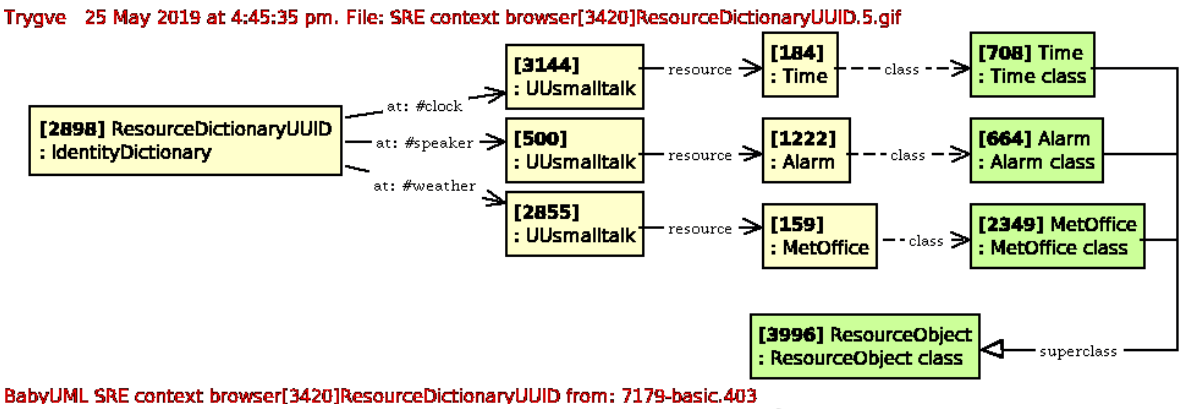
Notice that instances of UUID are not wrappers but objects that know how to access the features of their resource whatever its access mechanism.

> *Class UUID and its subclasses sustain the illusion that the Loke memory is a uniform universe of objects*
> *regardless of the diversity of their locality, access mechanisms and implementation details.*

BabyIDE maintains a cache of objects in a global Dictionary: [2898]ResourceDictionaryUUID. The objectDiagram in Figure 20 shows the structure of this Dictionary. The *Dictionary keys* are Universally_unique_identifiers and are visible to Ellen as icons in her resource window (Figure 7).[31] The *Dictionary values* represent RESTful servers and are instances of a UUID subclass.

[28] https://www.omg.org/spec/UML/2.5.1/PDF
[29] https://en.wikipedia.org/wiki/Representational_state_transfer
[30] UUID: https://en.wikipedia.org/wiki/Universally_unique_identifier
[31] For the purposes of the demo, they are simple Squeak Symbols; #clock, #speaker, and #weather).

Figure 20: Ellen's ResourceDictionarfyUUID
~~SRE Context browser[3420]ResourceDictionaryUUID.5.gif~~

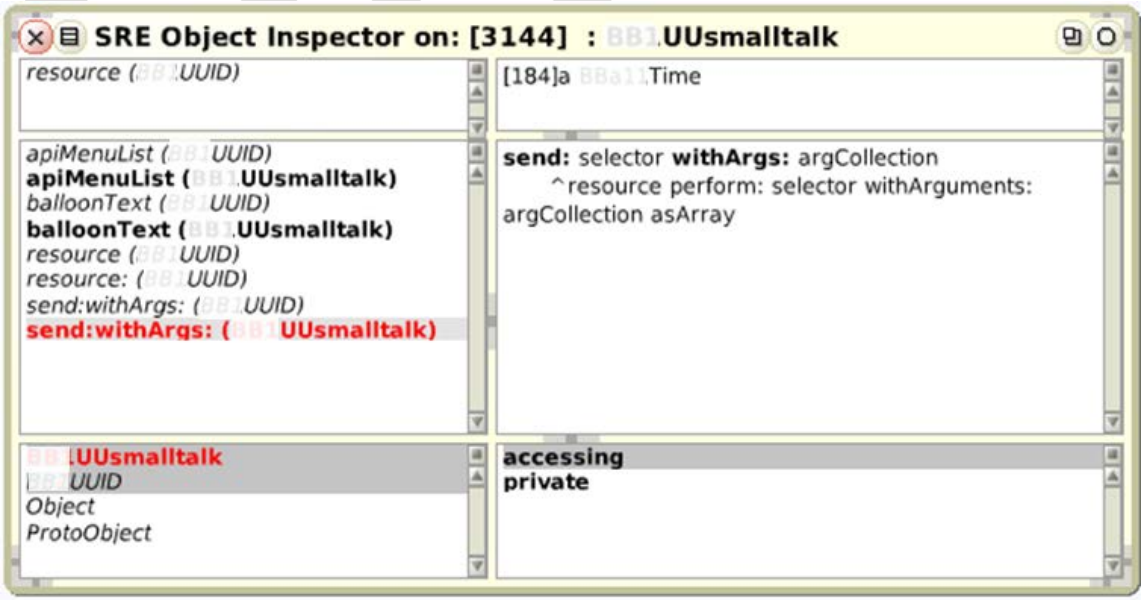


Ellen's personal objects are accessed through instances of UUSmalltalk, a subclass of UUID. The resource instance variable is the sole instance of a ResourceObject subclass.[32]. Shared objects will be identified by instances of other subclasses and their substructures may be different. In the future, 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.

D.2.Personal objects

Figure 21 shows an *SRE Object Inspector* on Ellen's #clock resource. The bottom-left pane shows its class with superclasses in a multi-select list. 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, accessing is selected. The middle-left pane shows their accessing methods. The essential send:withArgs: method is selected; its code is shown in the middle-right pane. The coalesced instance variables are in the upper-left pane. The upper-right pane shows the value, the [184]:Time object.

Figure 21: Ellen's Time identifier object[33]
~~SRE ObjectInspector on# [3144]-6.png~~



[32] BabyIDE has to be slightly modified if a ResourceObject class shall have more than one instance.
[33] The screen dump picture has been edited to cover class name prefixes not used in this article

We see from Figure 20 that like all personal objects, class Time is 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 hard coded in its resourceID method.
```

Ellen or her mentor program 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 "
```

D.3. A BabyIDE execution

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 (Figure 22).

Figure 22: The stack
SRE-traceRM-stack.png

```
1  [500] : UUsmalltalk>>send:withArgs: {wakeruppper 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
```

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 Context objects. BabyIDE leans heavily on this and other advanced features as will be seen in the following.

D.3.1. 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.

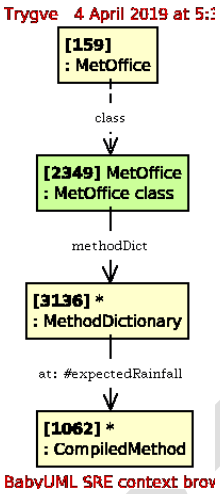
D.3.2. level 14 [3040] : InteractionRolePP >> startIn:

```
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 context diagram's TIMER symbol: [3040] :IntercationRolePP. 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 (Figure 21). In reality, objects delegate to their class to compile, store, and execute methods (Figure 23). Like all Squeak objects, object [159] has a link to its class: [2349]MetOffice. 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 goes across different abstractions.

Figure 23: An object's methods are stored in its class.
~~SRE context browser[359][159]a BBa11MetOffice.3.gif~~



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 (Figure 22)

```
Object subclass: #Context
    instanceVariableNames: 'roleMap'

Context subclass: #EllenAlarmCtx
    instanceVariableNames: ''
```

The class side declares the roles and their structure as a method that returns a Dictionary:

```
EllenAlarmCtx class>>roleStructure[34]
    ^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 moves an object into her Context. The declaration is later used by the compiler to find role names and permissible message paths. 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
```

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

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

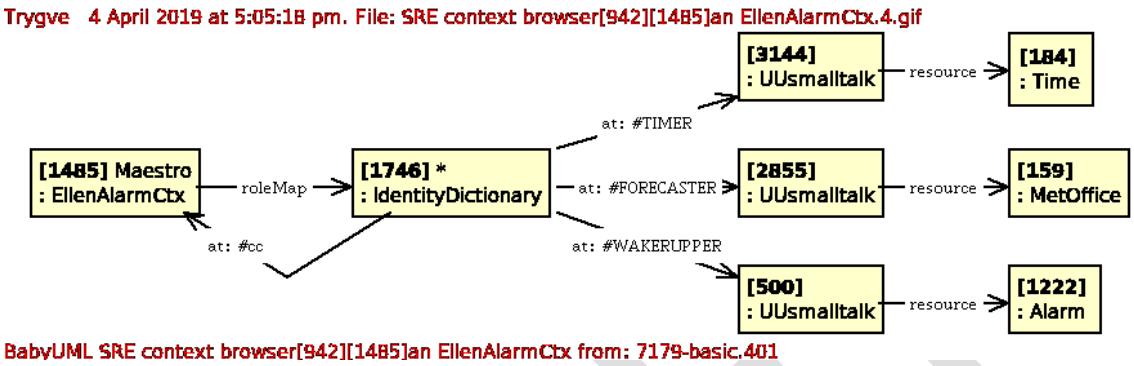
```
EllenAlarmCtx>>WAKERUPPER
    ^ResourceDictionaryUUID at: #speaker ifAbsent: [nil]
```

The roles with their default names and methods were created automatically when Ellen moved resource icons into her Context.

As part of its initialization, the Maestro executed Context>>remap to bind roles to objects. The result was the ephemeral object structure in Figure 24.

Figure 24: Maestro runtime object structure

~~SRE context browser[942][1485]an EllenAlarmCtx.4.gif~~



Note the "secret" role #cc. Every Context is initialized with this role; BabyIDE uses it on level 8 to find the current player of a given role.

D.3.3. level 13 [1485] : EllenAlarmCtx >> triggerInteractionFrom:with:

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

D.3.4. 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 a system operation: "
    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]]
```

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.

D.3.5. 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 system operation 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.

D.3.6. 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>
```

D.3.7. level 9 [1485] : EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:

We have digressed in levels 11, 10, and 9 to initialize the Maestro before we execute the inner (framed) block in this method from level 12. We leave the level of Squeak messages and methods and enter the higher abstraction level of roles and roleScripts (Figure 17). This higher level lets Ellen work in the role abstraction and protects her from the intricacies of Squeak with its classes. The role abstraction provides the foundation for Ellen's programming in section B: Novice Programming.

D.3.8. level 8 [1485] : EllenAlarmCtx >> to:send:withArgs:

```
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 Sqeak message"
            [^receiver perform: selector withArguments: argCollection asArray]]
```

All message sends in the Role abstraction are handled by this method. The messages are transformed into specialized messages to roles and objects, shared or personal.

> *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*

D.3.9. level 7 [3144] : UUsmalltalk(EllenAlarmCtxTIMER)>>waitTillMorning

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

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 level 14 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 no known class and a new home for roleScripts has to be found. Third, regular methods are executed in the context of the object. Here, there is no object and a new home for executing roleScripts has to be found.

> A side remark and confession: Being a one-man team, I had to suspend working with BabyIDE (fun) in order to write this article (a bore). The current implementation of BabyIDE compilation and execution appear to be unnecessarily complicated and a redesign is indicated.

The compilation of roleScripts

The roleScript compiler name space 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 converts Role names to regular Squeak code and then compiles the resulting code in the regular way.

Ellen's code:

```
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: {}
```

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 well-known Context>>to:send:withArgs: method 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.

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 and are very special classes with no instances, with no own methods, and Ellen can't see them.

The execution of roleScripts

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

D.3.10. level 6 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

D.3.11. level 5 [2855] : UUsmalltalk(EllenAlarmCtxFORECASTER)>>checkWeather

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

see level 7

D.3.12. level 4 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

D.3.13. level 3 [500] : UUsmalltalk(EllenAlarmCtxWAKERUPPER)>>wakeMe

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

see level 7

D.3.14. level 2 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

D.3.15. level 1 [500] : UUsmalltalk>>send:withArgs: {wakeruppper script wakeMe}

Finally, the stack was dumped on the Transcript before waiting until morning.

E. Epilogue

E.1. Further work

I see no reason to modify Loke's conceptual model at present, but its current implementation, BabyIDE, has been "programmed by inspiration" and leaves much to be desired. Below are of some of the things that need to be done with BabyIDE in both the short and the long term.

E.1.1. Refactoring BabyIDE

Restructure DCI runtime

Section *D.3: A BabyIDE execution* seems unnecessarily complicated and the implementation should be revised. A new RoleClass with subclasses can be made responsible for the static properties of the roles and their structure by merging the hidden classes that hold roleScripts with the class side of the Contexts. The instance side of the Contexts will remain responsible for managing the Interaction runtime. While a regular Squeak object responds to a received message by executing a method, a Context has an additional Role level that responds to a message by executing a role script. A new Context metaclass might simplify the handling of this added level.

Re-implement BabyIDE using BabyIDE

BabyIDE is the Squeak implementation of Loke with its IDE. The current program has grown "by inspiration". It should be redesigned and re-implemented using BabyIDE (BabyIDE was not available for its first implementation for obvious reasons). The architecture of *Appendix 2: ProkonPlan*, an Example in *Appendix 2: ProkonPlan, an Example* can inspire the architecture of a more readable version.

E.1.2. Extensions

Connect BabyID to the Net

Deploy the relevant Squeak communication packages to access shared objects. The first object will be our local weather service that offers a XML interface[35]. A subclass of UUID will transform it into a RESTful server. Ellen's smart alarm clock will be its first client.

Persistent information

The value of a program materializes when an end user applies it to create valuable data, the hardware and software are merely means to an end.

> *Users may tolerate an occasional program malfunction.*
> *They will never tolerate losing their data (including their personal programs).*

The target group for BabyIDE comprises millions of personal programmers. Each will create capital in the form of personal data. Ellen can use the full power of Squeak with its class library for roleScripts and many scripts may not survive the transition to a new version of Squeak. This is intolerable. We cannot expect that millions of personal programmer will have the motivation, time, and competence to convert all their

[35] https://api.met.no/weatherapi/locationforecast/1.9/documentation

programs. Also, their capital lies in the value of their information, not its representation. Information can therefore be stored in an independent repository such as a cloud. We will reuse an old and trusted scheme for representing the information content of an object:

> Whenever Ellen makes a change to an object that represents valuable information, its information content is stored in a record under its global identifier. The record contains a sequence of elements, one for each of the object's superclasses. In each superclass, a class method encodes its part of the information as an element that is tagged with information type and version number.
>
> The global identifier is used to retrieve the information record with its sequence of elements. For each element, its type is used to select the class that now represents its information. *This class is responsible for retrieving object information for all possible versions of the element.* A typical retrieval method is a long switch statement that binds version number to the appropriate decoder. The decoder is usually simple but can get quite complex in the rare cases when the class that represents the information has changed between store and retrieve time.

The result is that Ellen works with her programs and other information without ever worrying about where it is stored or how it is represented. Ellen can use the same repository for sharing her information with others.

Find and organize the objects

The Web with its emerging IoT is an extremely complex system with billions of independent objects, each running in their own process. This does not confuse this young and very modern man: "*If I don't find it in five minutes, it doesn't exist*".

The Things in Ellen's home are installed in her ResourceDictionary. She knows them well because she has bought them herself. From time to time, Ellen will need the additional functionality of a shared object. She will use a search engine to find it and install it in the same dictionary. The dictionary shows itself as icons on Ellen's desktop.

Error handling

Ellen's mental model is a model of objects that represent well-known, concrete things in her world. Her conceptual model is error-free and her Interaction tells the truth, the whole truth, and noting but the truth. Her simple model would become prohibitively complex if she were to include error handling in her program. The challenge to the Loke implementer is to let Ellen retain her illusion of an error free system and still capture the malfunctions that will occur from time to time in the concrete system. In section C4, I wrote that 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 wakes up at 10 o'clock in the morning in beautiful sunshine with an error message: *I didn't wake you because something went wrong*.

Create an alpha version of Loke

An alpha version shall be available to anybody who has the courage and interest to be on the bleeding edge of the development. This version will be based on the re-implemented BabyIDE described above. Its opening screen will look something like Figure 7 and the many Squeak facilities will be hidden in the manner of Etoys[36]. The re-implemented BayIDE will run in an old version of Squeak [37] and must be ported to the latest version before publication. There are many differences between the old and the latest versions.

[36] http://www.squeakland.org/
[37] Squeak version 3.10.2

For example, there is a new compiler so that the compilation of role scripts has to be redesigned. Other problems are likely to surface during the porting.

The alpha version will not be secure. There will be vulnerabilities in the underlying operating system and also in the Loke implementation itself. It is sufficient to note that it will be using the HTML 5 communication protocol, a protocol that does not appear to be designed for maximum security.

Extend project collaboration

The Loke project needs to work within a real environment. The project should therefore aim to collaborate with another free and open source IoT project.

E.1.3. We need a paradigm shift

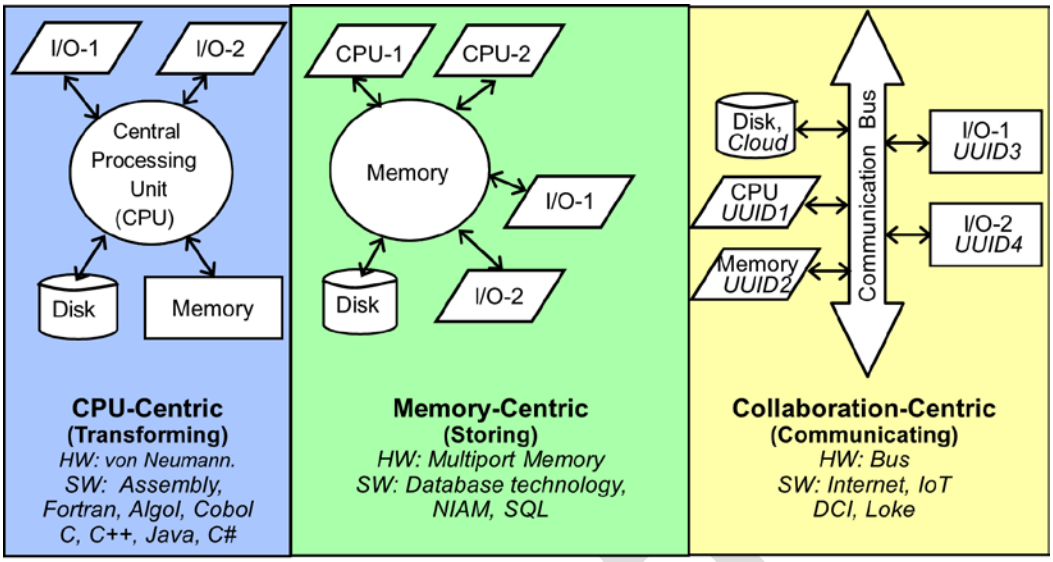
The history of Western astronomy shows a series of paradigm shifts from the geocentric paradigm with its bizarre epicycles via the heliocentric to the current distributed paradigm with its chunks of mass connected by gravity. What appeared as essential complexity in one paradigm is easily resolved in the next.

It is tempting to look for a similar paradigm shift for computing. Mainstream programming has based its theory and practice on a CPU-centric paradigm exemplified by the von Neumann machine of 1948. The transition to a Memory-centric paradigm is the next paradigm shift.

It is time to realize that these two paradigms no longer meet our 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 realizing their sophisticated requirements.

Computers can transform, store, and communicate data with the software always lagging behind the hardware. (Figure 25) 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 Collaboration-centric paradigm is that computers are primarily used to exchange messages with other computers; computers collaborate to achieve a common goal.

Figure 25. The three capabilities of computing
ComputerArchitectures-3.png (#253)



It is time to heed Tony Hoare's plea for simplicity and achieve a better way of separating concerns. Mainstream programming should move on to the *collaboration-centric paradigm* exemplified by the object computer that is the foundation for this article.

The collaboration-centric paradigm has been on the horizon for many years. We first met it in Prokon's idea of distributed computers (Reenskaug, 1977). A current example is *Service-Oriented Architectures (SOA)* that, in essence, is collaboration-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 [38]. And of course, DCI and the IoT itself are, by definition, collaboration-centric.

E.1.4. 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 (Kay, 1972), should cater for all its owner's computing needs. The computer was to be self-contained and include the hardware and software needed to run a Loke personal computer. I have dropped this idea. Potential users like Ellen depends on her smartphone in her daily life. She uses it for many purposes ranging from paying her groceries to finding the meaning of the word *astrobleme*. It will not be possible to reprogram all these services on a dedicated Loke device. The first Loke device must therefore be like an existing smart device augmented with the Loke alpha version.

Ellen owns many devices, and she wants to run her Loke computer on all of them. The Loke computer must 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 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.[39]

[38] https://en.wikipedia.org/wiki/Distributed_computing
[39] https://www.parallels.com/blogs/ras/remote-application-overview/

Ellen will use her Loke computer to access any available program on the net, including her personal Loke; it 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.[40]

The icons on the Loke desktop represent the Loke IDE and other executable resources such as drawing programs, word processors, web browsers, etc. Ellen can use the Loke IDE to create and deploy new resource objects that can be either personal or shared. Ellen's personal data will be moved out of the device and into a shared repository such as a cloud.

The Loke computer only needs a minimal operating system such as a microkernel [41] or Squeak itself. This will significantly reduce it vulnerability to malicious attacks as will the security barriers between kiosk and remote applications.

The Loke computer will be rendered useless if it loses contact with the Net. It will therefore be equipped with extensive caching so that it can be used off-line.

[40] https://en.wikipedia.org/wiki/Kiosk_software#Security_features
[41] https://en.wikipedia.org/wiki/Microkernel

E.2. Related work

I am an octogenarian. 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 library. I am only too aware that I have missed essential initiatives and offer my sincere apologies for not having discovered them.

TBD

Objectteams

E.2.1. Disruptive initiatives

Interplanetary File System (IPFS)
https://hackernoon.com/a-beginners-guide-to-ipfs-20673fedd3f

Microservices

E.2.2. trygve language

The DCI programming paradigm has been reified by James O.Coplien (Cope) in a research language he calls **trygve:**[42].

The **trygve** language allows you to think about your program in terms of mental models of some phenomenon that you bring to the table. It's called object-oriented programming, so we're going to talk about classes as a supporting cast rather than as the star players. Here, objects are the stars of the show. We talk about objects in terms of the names our compressed mental models give them as they interact to solve some problem. Those names are Roles. Any given Role is just a name but at the same time is much more. The Role "fireman" is just a name for some human being in a particular Context, but it also elicits a host of associated behaviors. Those behaviors that are germane to the Role itself, without much regard for the Role-player, are called *scripts* in **trygve** (computer scientists call them *methods*) — little recipes for doing very small tasks.

And **trygve** departs from Java in yet more fundamental ways. Java tried to be a pure OO language by outlawing global functions, but that is a simplistic hope at best. It ended being only a class-oriented programming language. Like most languages of its kind it has many features to finesse class relationships. These features encourage class-oriented thinking, overuse of inheritance, and programmer convenience over end-user mental models. So you will find neither friends, or static objects, or the concept of super, nor the protected access property in **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 right entities (objects) happens on the screen, and Model-View Controller (MVC) has been designed as the bridge between end user and computer in that regard.

[42] http://fulloo.info/Documents/trygve/trygve1.html

MVC and **trygve** can powerfully be combined to provide the most expressive links between the end user and the machine.

The source code of a **trygve** program may contain the following elements[43]:

```
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 code differs from Java with its two special keywords: **context** and **role**, words that have roughly the same meaning as in section B.2. The **trygve** source code is a complete text that declares the program as a whole. This makes it possible to analyze the program automatically and support functionality such as its typing system. BabyIDE and Loke are based on Squeak. They lack the notion of a source code in closed form and can only provide dynamic typing. In **trygve**, the roles form an unordered set, while in BabyIDE and Loke they form a directed graph that declares role visibility. The graph is also part of the programming language in BabyIDE. (**Error! Reference source not found.**).

The DCI paradigm has been reified in several mainstream languages such as Ruby, C++, Scala, and Java. Also in Rune Funch's Marvin DCI language. [44]

Stephan Hermann's ObjectTeams [45]have a concept similar to the DCI role and share many of the properties of DCI. ObjectTeams creates a role by wrapping the role-player with a separate object, thus having several identities for an entity that is conceptually a single object. This does not cause any problems in many cases, but this object schizophrenia may cause malfunctioning that is hard to track down.[46]

----------Fragments/ideas:----------

For kids: EToys, Scratch, mainstream language (Python) pre-processors?

School curriculums to make everybody computer literate.

[43] Example code in http://fulloo.info/Examples/TrygveExamples/
[44] http://fulloo.info/Examples/
[45] http://www.eclipse.org/objectteams/

[46] 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

E.2.3. Actor Model

Carl Hewitt defined the *Actor Model* in 1973 as a mathematical theory that treats "Actors" as the universal primitives of concurrent digital computation.
Carl Hewitt defined the *Actor Model* in 1973 an actor is a process like nodes in a neural network
Imperative actors send an RPC destined for an actor with known interface, type,
Relevant model for BabyIDE/**IoT?**
https://www.infoq.com/articles/reactive-cloud-actors
https://dzone.com/articles/distributed-systems-done-right-embracing-the-actor-model

DCI: sequential (synchronous).
Actors is parallel; not studied here. Harder to read code. too abstract for Ellen.

E.3. Summary and Conclusion

https://www.home-assistant.io/

I postulate that every thing and every person will be connected through an emerging communication network such as an IoT in a smart home. I further postulate that the connected things be represented by objects that can be accessed over a network through their provided message interfaces.

Objects are entities that encapsulate state and behavior and have a unique and immutable identity. The encapsulation separates the outside of the object from its inside with a barrier that theoretically ensures security and privacy. The outside provides a message interface to a possibly evil environment. The inside is invisible from the outside and handles incoming messages in different ways depending on the nature of the object. The only way for a thing like a thermometer to worm its way into the heart of the corporate information system is through a chain of provided interfaces.

I claim that the Loke object encapsulation *provides a promising security and privacy mechanism in the war against malware.*

My focus is on the human use of resource objects. My target group consists of people who need to go beyond ready-made apps and create their own programs. I call them personal programmers: *While a professional Programmer is a highly trained expert, a personal programmer has other interests and values simplicity and convenience over programming sophistication.*

Ellen played with Lego bricks in her childhood and clicked selected bricks together to form what's called a *Lego project*. Ellen now plays with Loke objects and links them together with communication paths to form what's called a *Loke Context*. While Lego bricks are dumb pieces of plastic, Loke objects are smart and respond to messages. Ellen augments the behavior of her own and other people's objects with scripts that makes them collaborate to achieve her goals. Just as Ellen can click Lego projects together to form larger projects, she can link Loke Contexts together to form larger Contexts: The idea of a Loke Context is recursive.

I have implemented a first version of Loke. It sustains two essential illusions: One is that objects look the same wherever they are situated and whatever their implementation technology. The other is that all messages look the same independent of their transmission technology and their communication protocol.

One purpose of the implementation is to demonstrate the effectiveness and intuitiveness of its model and programming interface. Ellen feels that Loke is intuitive because it is close to her innate mindset. Its objects represent well-known things in her environment and appear as visible and tangible icons on her screen just as the resources in her smartphone. I have created a demonstration of how Ellen can program a smart alarm clock.[47] Informal demonstrations indicate that people familiar with the web and smart devices immediately grasp the idea of objects that represent concrete things in their environment. They also accept the idea of instructing objects to interact to serve their needs.

Non-programmers watching the demonstration were creative in identifying everyday opportunities for IoT and personal programming. They seemed to understand the idea of programming by composition and found it easy to identify the things that must play roles in their solutions. They also described informally what the objects should do to reach their goals. The concrete demonstration was an essential key to their understanding and I am convinced that a verbal exposition could not reach their minds in the same way.

I demonstrated Loke programming to a farmer (my daughter). She had been using computers for many years but has never written a program. We were brainstorming a number of possible applications of Loke on her farm. She came up with many ideas, identified the participating objects, and figured out what they should do. One of the farm activities is to raise bull calves for meat production. She inspects the cowshed several times a day but it is sometimes hard to see if a certain calf is sickly or only resting. Her solution was to monitor the automatic feeding system for every calf and to warn her if a calf was off his feed. She ended the session 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 more like setting up a playlist in a music server than programming in the conventional sense of the word. There is no self-contained source file and no concrete program; yet Loke can do anything a network of communicating computers can do. Ellen's way of programming heralds a lifting of the universe of discourse from the von Neumann machine to the universe of executable objects available through Loke, her personal object computer.

Unknown to people watching the demonstration, they actually observed the creation of a Loke program through its interactive GUI that takes the place of a programming language. Every time Ellen creates a program through the GUI, she also intuitively builds her mental model of the Loke object computer. Her experiences will always be consistent since they reflect the semantics of Loke.

Interestingly, conventional programmers who attended the demonstration and who were used to working with code in textual form didn't see the point of Loke. The Loke way of programming felt disruptive and they could easily solve the same problems with textual code in their favorite language. But then, they don't need Loke and do not belong to its target group.

I claim that Loke is what it set out to be:
A simple and safe path to the art of programming for laypeople
that is intuitive because it builds on innate human traits.

[47] A short video illustrates the novice IDE: http://folk.uio.no/trygver/themes/Personal/TrygveReenskaug-PersonalProgramming(AVCHD.H264.1440x1080p24).mp4

I singled out Anton as another member of the Loke target group and who had advanced from novice to expert through creating new programs to satisfy new demands. He had become acquainted with more resource objects as and when he needed them. He has also extended his repertoire of Loke programming facilities. For example, he now writes role scripts as texts to circumvent the limitations of the menu-driven approach. Also, where Ellen selected objects by dragging them into the Context, Anton selects them programmatically.

Anton may take a further step towards becoming an expert personal programmer by attacking problems outside the realm of IoT, using the Data, Context, Interaction (DCI) programming paradigm that is at the core of all Loke programming.

I claim that Loke is a computer for general personal support
much like Alan Kay's Dynabook[48]
because it has all the capabilities of a computer packaged for non-professional owners.

My Loke implementation is a non-intrusive extension of Squeak, a variant of Smalltalk. Squeak is a universe of objects and nothing but objects. The objects represent classes, compilers, methods, messages, stack frames, etc. They are all regular Squeak objects; i.e. instances of regular Squeak classes[49]. Like Squeak, a Loke execution is a flow of messages through an ensemble of objects that are selected at runtime.

My implementation is an executable, multidimensional, conceptual model of Loke that uses Squeak as its medium. A reader of the model can explore its objects and their relationships. A reader can also observe the execution of existing and new programs in order to explore Loke's dynamic properties. Loke qua conceptual model is completed. The first Loke implementation can guide an implementer of a new version of Loke, for example in a different development environment such as Pharo[50]

I claim that the Loke implementation
is a conceptual model of Loke for inspection and exploration
because it embodies the whole of Loke in executable form.

It is still early days for the Loke implementation qua a programming environment. It needs he addition of many facilities while retaining its simplicity and intuitiveness. For example: Loke is not connected to the Net and facilities for publishing new objects for personal or shared use are missing. Of special interest is to refine Loke's inherent security mechanism through future implementations. The current implementation rests on a popular operating system and is therefore inherently insecure. A better approach could be to implement Loke directly on dedicated hardware; a Loke machine that provides one or more message interfaces to its environment and that uses resource objects available on the net. A Loke would then appear as both client and server in an IoT client-server architecture.

The Loke development has been a one-man project since I started it in 2015. The advantage of being alone has been that it has been easy drop one line of research at any time and start afresh. The disadvantage is that it is limited what a single person can do. I

[48] https://en.wikipedia.org/wiki/Dynabook
[49] Smalltalk (and Squeak) is sometimes referred to as a programming language. This is a misleading understatement. The "language" is essentially one of many possible languages for declaring methods, it lacks the concept of a program, and its many innate capabilities for introspection and reflection are ignored. The Loke implementation builds heavily on these deep capabilities and uses class-oriented programming only sparingly.
[50] https://pharo.org/

am an octogenarian and lack the necessary energy for bringing Loke to fruition. The time is ripe to scale 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. May be that in the future, it will transpire that Loke can be seen as a conceptual model of the human use of connected things.

E.4. Acknowledgements

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 had both been following our separate paths when searching for a common truth we both have felt must be out there somewhere. On Aug 28, 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[51]. I spread the good news to a 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.

My sincere thanks to the active community centered around the object-composition@googlegroups.com mailing list for their contributions to the DCI paradigm and its dissemination.

The BabyIDE implementation rests heavily on Traits. My sincere thanks to Nathanael Schärli, Stéphane Ducasse, Andrew Black, and Adrian Lienhard for providing this very pow-erful extension of the Squeak class paradigm.

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 .The value of their disruptive ideas cannot be overestimated.

[51] BabyIDE works under Squeak version 3.10.2. It is not easily converted to later, more complicated versions.

E.5. References

Needs to be pruned.

Ahmad, M., 2014. *Reliability Models for the Internet of Things: A Paradigm Shift.* Naples, Italy, IEEE.

Alan Kay, A. G., 1977. Personal Dynamic Media. *Computer*, Issue March 1977, pp. 31-41.

Alaya, M. B., 2015. *Towards interoperability, self-management, and scalability for machine-to-machine systems. Networking and Internet Architecture [cs,* s.l.: Universite Toulouse III Paul Sabatier,.

ANSI, 1998. *Programming Language Smalltalk.* s.l.:InterNational Committee for Information Technology Standards (formerly NCITS), 01/01/1998.

Bluemke Ilona, S. A., 2015. *Experiences with DCI Pattern.* s.l., Springer, pp. 87-96.

Buxton, J. & B. Randell, e., 1969. *Software Engineering Techniques.* Rome: NATO Science Committee.

Choi, J. K. & Kim, H. J., 2016. *Analysis of Digital Data Technologies Toward Future Data Eco-Society,* s.l.: International Telecommunication Union.

Coplien, J. O., 2010. *Lean Architecture for Agile Software Development.* Chisester, UK: John Wiley & Sons Ltd.

DCI-list, 2017. *DCI - Data Context Interaction.* [Internett]
Available at: http://fulloo.info/

Dijkstra, E., 1972. The Humble Programmer (Turing Award Lecture). *Communications of the ACM*, October, pp. 859-866.

Dijkstra, E. W., 1969. 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.

Diskin Z., K. H. L. M., 2028. *Multiple Model Synchronization with Multiary Delta Lenses.* s.l., Springer, Cham, pp. 21-37.

Ducasse, S., 2005. *Squeak: Learn Programming with Robots.* s.l.:Apress.

Emery, F. & Thorsrud, E., 1976. *Democracy at Work: The Report of the Norwegian Industrial Democracy Program (International Series on the Quality of Working Life).* s.l.:s.n.

ESUG, 1995. *VisualWorks Advanced Tools,* s.l.: European Smalltalk User Group (ESUG).

Galas, C. & Freudenberg, R., 2010. *Learning with Squeak Etoys,* s.l.: s.n.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns. Elements of Reusable Object-Oriented Software1994.* s.l.:Addison-Wesley.

Goldberg, A. & Robson, D., 1983. *Smalltalk-80, the language and its implementation.* s.l.:Addison-Wesley.

Green, T., 1995. Noddy's Guide to Visual Programming. *The British Computer Society • Human-Computer Interaction Group "Interfaces"*.

Hameed, S., Kahn, F. I. & Hameed, B., 2019. Understanding Security Requirements and Challenges in Internet of Things (IoT): A Review. *Journal of Computer Networks and Communications*, Volum 2019.

Hoare, C., 1981. The Emperor's Old Clothes. *Communications of the ACM*, February, pp. 859-866.

IFIP, 1966. *IFIP-ICC Vocabulary of information processing.* Amsterdam: North-Holland Publishing Company.

ISO, 2009. *ISO 80000-1: Quantities and units. Part 1: General (1st ed.).*, Switzerland: ISO (the International Organization for Standardization.

Kay, A., 1972. A Personal Computer for Children of All Ages. *ACM '72 Proceedings of the ACM annual conference - Volume 1*, 1 August, p. Article No 1.

Kay, A., 1989. *User Interface: A Personal View.* [Internett]
Available at: http://www.vpri.org/pdf/hc_user_interface.pdf

Kay, A., 1993. The Early History of Smalltalk. *ACM SIGPLAN Notices archive*, March, pp. 69-95.

Kelly, K., 2008. *The next 5,000 days of the web.* [Internett]
Available at:
https://www.ted.com/talks/kevin_kelly_on_the_next_5_000_days_of_the_web#t-8111

Reenskaug, T., 1973. *Administrative control in the shipyard.* Tokyo, ICCAS conference. Also at http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf.

Reenskaug, T., 1977. Prokon/Plan - a Modelling Tool for project Planning and Control. *IFIP Congress*, pp. 717--721.

Reenskaug, T., 1978. *MVC, XeroxX PARC 1978-79.* [Internett]
Available at: http://folk.uio.no/trygver/themes/mvc/mvc-index.html

Reenskaug, T., 1988. *A Methodology for the Design and Description of Complex, Object-Oriented Systems*, Oslo: Central Institute for Industrial Research, Oslo.

Reenskaug, T., 1996. *Working With Objects. The OORAM Software Engineering Method.*. Greenwich: Manning.

Reenskaug, T., 2006. Expert' voice: The BabyUML discipline of programming. *Software and System Modeling*, 5(1), pp. 3-12.

Reenskaug, T., 2007. Expert Commentary: The Case for Readable Code.. I: *Computer Software Engineering Research.* New York: Nova Science Publishers, pp. 3-8.

Roe, D., 2018. *7 Big Problems with the Internet of Things.* [Internett]
Available at: https://www.cmswire.com/cms/internet-of-things/7-big-problems-with-the-internet-of-things-024571.php
[Funnet 06 05 2018].

Squeak, c., 2018. *Monticello.* [Internett]
Available at: http://wiki.squeak.org/squeak/1287

Valdecantos, H. A., 2016. *An empirical study on code comprehension: DCI compared to OO,* s.l.: Rochester Institute of Technology.

Wiegers, K. E., 2002. *Seven Truths About Peer Reviews.* [Online]
Available at: http://www.processimpact.com/articles/seven_truths.html

Wirfs-Brock, R. J. & Johnson, R. E., 1990. Surveying current research in object-oriented design.. *Comm. ACM,* 33(September 1990), p. 113.

Wirfs-Brock, R. & McKean, A., 2003. *Object Design. Roles, Responsibilities, and Collaborations.*. Boston, MA: Addison-Wesley.

F. Appendix 1: BabyIDE User Manual

BabyIDE is an non-intrusive extension of the Squeak universe of objects that supports the DCI separation of concerns. This separation is achieved by viewing the program in separate projections (Figure 4), each projection tells part of the story; all projections taken together tell the whole story. BabyIDE provides specialized browsers for the Data, Context, and Interaction kinds of projections.

Ellen uses a variant of BabyIDE that is scaled down for the novice (Figure 7). Expert programmers use the full BabyIDE that is described in this section and exemplified in section G. Appendix 2: ProkonPlan, an Example.

BabyIDE has different browsers for different kinds of projections. The browsers are described in more detail below. We stress that a programmer may only use a selection of these browsers. Code in other projections is then generated automatically as we have seen in Ellen's IDE.

BabyIDE makes the program architecture visible and tangible with browsers for each projection. The BabyIDE browsers support the DCI separation of concerns so the work done on a projection in one browser is almost independent of work done on other projections in other browsers. BabyIDE is modeless; each browser carries its own state so that work in a browser can be suspended and resumed at will.

- ***The Interaction browser*** is for editing the Interaction diagram with its roles and the links between them. It also supports creating and editing the roleScripts that drive the peer to peer communication when the system performs a system operation.
- ***The Context Class browser*** is for working with the class of the context objects. Its provided interface includes the system operations that are realized by the Context. This class also includes the essential methods that bind Roles to objects dynamically during the execution of a system operation.
- ***The Data Class browser*** is for working with personal Squeak classes.

I use Ellen's code as an example throughout.

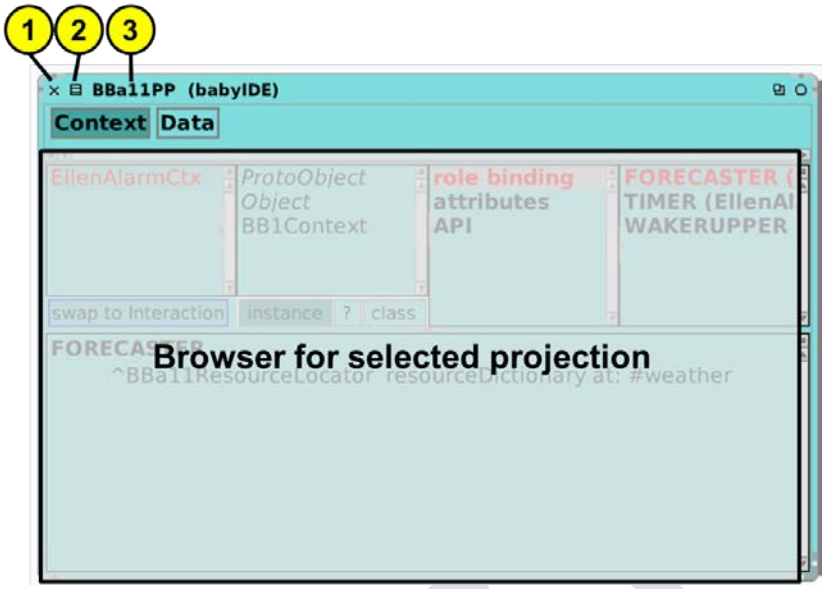
F.1. The shared heading of all BabyIDE browsers

The headings of all BabyIDE browsers are (Figure 26):

1. This button closes the window.
2. A menu button, its commands are
   - remove application *Remove this application from the system.*
   - change application *Change the application that is handled by this IDE.*
   - new projection: *Add a new projection to this application.* (Not implemented)
   - printHtml for this App: *Export a textual projection of this application as a read-only HTML file.*[52]
3. The application is Ellen's Smart Alarm, *BBa11PP*. (The obscure name prefix is chosen to get around the lack of name spaces in Squeak).

[52] Example in http://folk.uio.no/trygver/assets/BBa11PPEllen/readableVersion.html.

Figure 26: BabyIDE heading
BB11PP-Window-annotated.png



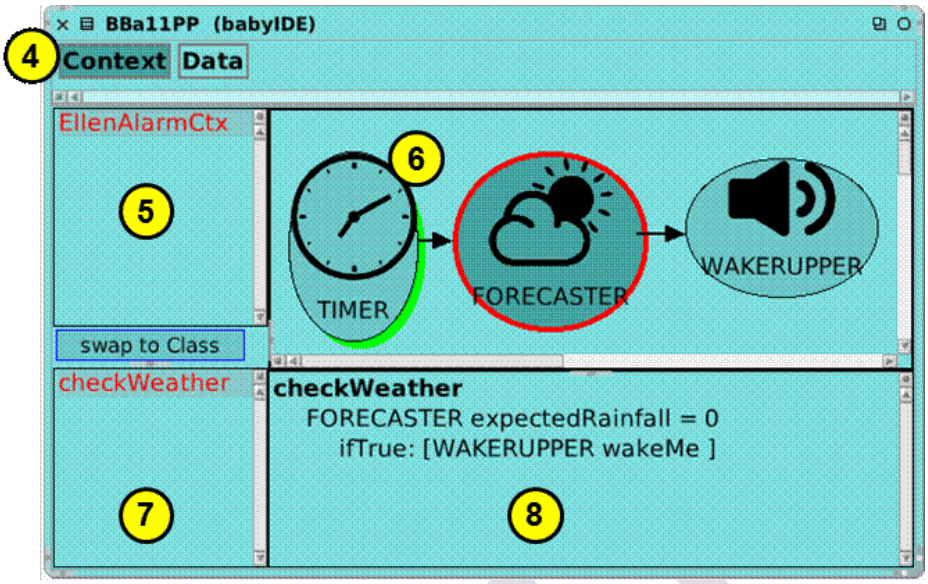
New projections are added as needed. Each projection is edited in an appropriate browser.

F.2. The Interaction Browser

The Interaction browser (Figure 27) is where the programmer specifies how a DCI program realizes a System Operation within a network of communicating objects. The code in this projection answers three critical questions for each System Operation:

- *What are the objects?* The roles identify the participating objects; the actual objects are selected at runtime by the Role binding methods in the Context Class. A role names an object and does not know the object's implementation, e.g. as an instance of a class. In BabyIDE/Novice, Ellen chooses the objects directly and the role binding methods are generated automatically. (Figure 28 pane 9).
- *How are they interlinked?* The programmer answers this question by defining a network of roles in the Interaction Diagram.
- *What do they do?* The browser answers this question by augmenting selected Roles with *roleScripts* (Figure 27 pane 8). The BabyIDE runtime system creates the illusion that these scripts are parts of the roleplaying objects while they are needed at run-time. In reality, the objects are unchanged while the extra behavior is added by the runtime system.

Figure 27: The Interaction Browser.
BB11PP-ContextInteraction-annotated.png



The panes of the Interaction Browser (Figure 27) are:

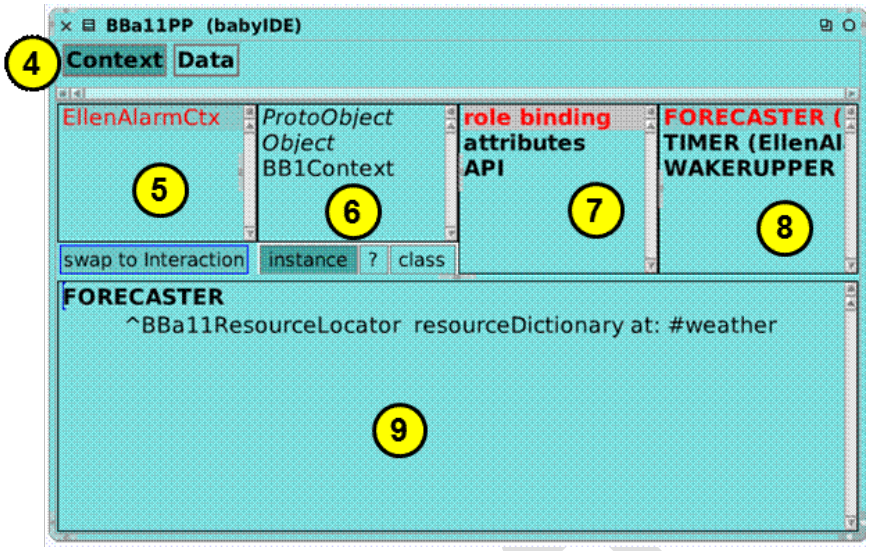
4. The projection is named *Context*.
5. A list of contexts, one for each system operation. *EllenAlarmCtx* is selected. A button toggles between the Context class and Interaction browsers.
6. The Interaction diagram where the programmer works with roles (select, link, add, remove, and rename). The FORECASTER role is selected.
7. A list of FORECASTER roleScripts, FORECASTER>>checkWeather is selected.
8. An editor for the selected roleScript. roleScripting is discussed in (section B.2.2). (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.)

Figure 31 shows a more complex example.

F.3. The Context Class browser

A Context object forms the environment for the execution of roleScripts. The properties of the Context are edited in a Context Class Browser. A *swap to class/swap to Interaction* button toggles between the two browsers.

Figure 28: The Context Class browser.
~~BB11PP-ContextClass-annotated.gif~~



The panes of the Context class browser (Figure 28) are:

4. The projection is named *Context*.
5. A list of Contexts, EllenAlarmCtx is selected. A button toggles between the *Context class* and *Interaction* browsers.
6. The superclasses of the selected class are shown in a multi-select list that acts as a presentation filter.
7. The method categories of the selected class and any selected superclasses are shown in a multi-select list. This is a presentation filter; only methods belonging to the selected class, the selected superclasses and the selected method categories are shown. The role binding category is selected. By convention, the API category is reserved for the provided message interface of this Context, i.e., the system operations it implements.
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 an object. The role binding methods are always executed together as one atomic operation to ensure consistency. Ellen generated these methods automatically when she moved Data objects into her Context).
9. A code pane for editing the source code of the selected method. This is a text pane, and it lets the programmer to write code in Squeak's default language for methods.

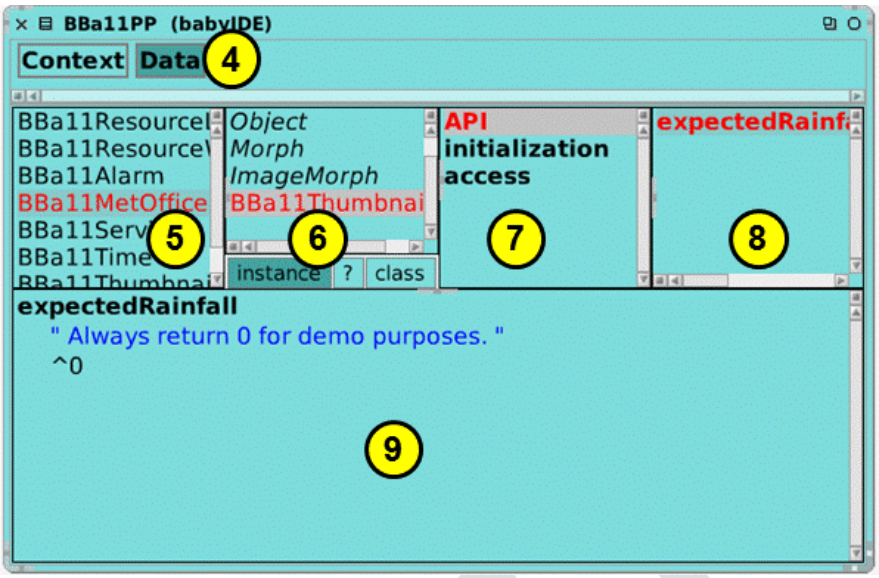
F.4. The Data Class browser

Any object with a globally unique identity and with a known message interface can play a role in a Context. Available objects are found in a repository (Figure 7). This means that an instance of any Squeak class can play a role, including an instance of a Context class. The Data class browser is used to define some of these classes, other classes are defined elsewhere.

A role in a Context names an object with its identity and interface. The role does not know the object's implementation, e.g. as an instance of a class.

In the BabyIDE Data projection, we edit the declaration of classes that are specific for the current application in the Data class browser (Figure 29).

Figure 29: The Data class browser.
(BB11PP-Data-annotated.gif)



The panes in the Data class browser (Figure 29) are:

4. The selected projection is *Data*.

5. A class list showing the personal Data classes. BBa11MetOffice is selected. This a dummy that simulates an Internet service. Internet objects are not handled in this browser, but are handled in a separate repository.

6. The superclasses of the selected class are shown in a multi-select list that acts as a presentation filter.

7. The method categories of the selected class and any selected superclasses are shown in a multi-select list. This is a presentation filter; only methods belonging to the selected class and the selected superclasses are shown. The API category is selected. By convention, its methods form the external interface of this class.

8. A list of methods in the selected method categories. The expectedRainfall-method is selected.

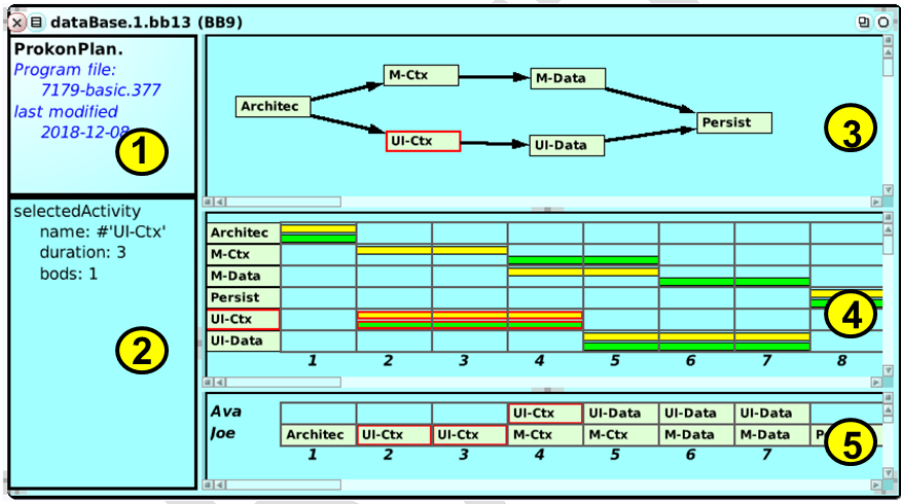
9. A code pane editing the source code of the selected method.

G. Appendix 2: ProkonPlan, an Example

Anton's plans were in the form of activity networks where an activity represents a task that needs to be done. An activity has a certain duration, it cannot start before all its predecessor activities are completed, and it must end before any of its successor activities can start. Different managers are responsible for different activities. Anton works with a personal fragment of the overall plan. His collaborators publish planning objects that Anton connects to and Anton publishes one or more planning 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 a distributed environment (Figure 14).

Figure 30: ProkonPlan user interface.
~~Prokon-3-annotated.png~~



I used BabyIDE to program this application as an example of MVC/DCI programming. Its user interface has a number of panes as shown in Figure 30:

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).
3. Shows the activity dependency graph that connects predecessor and successor activities.
4. Shows the earliest and latest start and finish times that 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 earliest and the bottom, green bars show the latest times (week numbers).
5. Shows the results of an automatic resource allocation.

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 in all:

- *Model-Data projection*. The user experiences the program's user interface as an extension of their mind. This "magic" of MVC (Figure 3) is achieved by faithfully

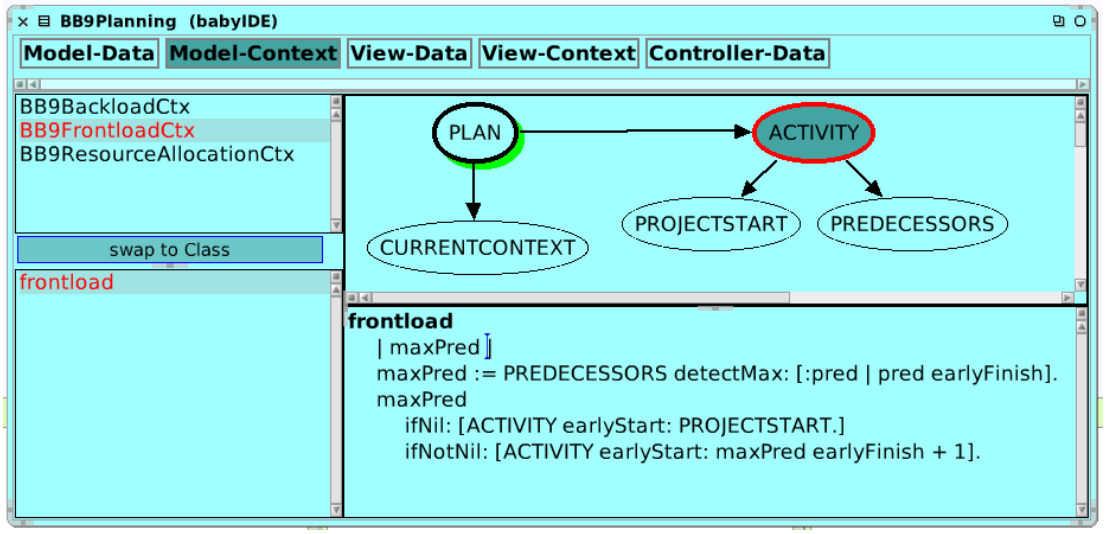
reflecting the user's mental model in the *Model Data*. The classes are DBase, Model, Activity, Dependency, Resource.

- *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 FrontloadCtx, BackloadCtx, ResourceAllocationCtx.
- *View-Data projection.* The "magic" of MVC (Figure 3) is partially achieved by using well known graphic languages for the Views so that they are readily intuited by the user. The views thus bridge the gap between the human mind and the Model. The classes are DependencyView, ActivitySymbol, DependencyLine, ActivityTextView, 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 with its Views and it is responsible for the selection mechanism. I only use DCI when it leads to simpler and more 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 text projection of ProkonPlan has been printed on file[53]. It clearly illustrates how the MVC/DCI separation of concerns leads to almost independent projections where each projection can be separately written, reviewed, and tested. The exception is the selection mechanism that is implemented in class-oriented code.

The program was implemented using BabyIDE (Appendix 1: BabyIDE). The projection names are shown in the top line of the screen dump in (Figure 31). *Model-Context* is selected. The Context diagram shows the participating objects and the source pane shows the roleScript for frontloading one activity at the time.

Figure 31: BabyIDE for ProkonPlan screen dump
~~BB9Planning-BabyIDE.png~~



[53] http://folk.uio.no/trygver/assets/BB9Planning/BB9Planning-listing/readableVersion.html
http://folk.uio.no/trygver/assets/BB9Planning/BB9Planning-listing/printableVersion.html

The loop for traversing all the activities is in the PLAN roleScript

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

The CURRENTCONTEXT object 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".
```

A final remark:
In Figure 30, pane 5 we see that the UI-Ctx activity is to be performed by Joe in week 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 code this particular algorithm to get an acceptable plan. This is feasible because the code is wholly contained within a single Context, the ResourceAllocationCtx.