Designing and Programming Malleable Software

Philip Tchernavskij

université PARIS-SACLAY

NNT : 2019SACLS499

Thèse de doctorat

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

Designing and Programming Malleable Software

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique (Interaction Homme-Machine)

Thèse présentée et soutenue à Orsay, le 03/12/2019, par

PHILIP TCHERNAVSKIJ

Composition du Jury :

Jean-Daniel Fekete Directeur de Recherche, INRIA Unité de Recherche Saclay	Président
Stéphane Conversy Professeur, ENAC-LII et Université de Toulouse	Rapporteur
Nicolai Marquardt Associate Professor, University College London	Rapporteur
Myriam Lewkowicz Professeur, Université de Technologie de Troyes	Examinatrice
Jutta Treviranus Professor, OCAD University	Examinatrice
Michel Beaudouin-Lafon Professeur certifié, Université Paris-Sud	Directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Concevoir et programmer des logiciels malléables

Mots clés : Interaction Homme-Machine, Modélisation Conceptuelle, Ingénierie des Systèmes Interactifs

Résumé : Les besoins des utilisateurs en matière de fonctionnalités et d'interfaces logicielles sont variés et changeants. Mon objectif est de permettre aux utilisateurs eux-mêmes de facilement modifier ou faire modifier leur logiciel en fonction de l'évolution de leurs besoins. Toutefois, à mon avis, les approches actuelles ne traitent pas cette question de façon adéquate: L'ingénierie logicielle favorise la flexibilité du code mais, dans la pratique, cela n'aide pas les utilisateurs finaux à apporter des changements à leurs logiciels. Les systèmes permettant à l'utilisateur de programmer en direct ("live programming") ou de modifier le code du logiciel ("end-user programming") permettent aux utilisateurs de personnaliser les interfaces de leur logiciel en accédant et modifiant le code source. J'adopte une approche différente, qui cherche à maximiser les modifications qui peuvent être faites à travers des interactions habituelles, par exemple la manipulation directe d'éléments d'interface. J'appelle cette approche la malléabilité logicielle. Pour comprendre les besoins des utilisateurs et les obstacles à la modification des logiciels interactifs, j'étudie comment les logiciels actuels sont produits, maintenus, adoptés et appropriés dans un réseau de communautés travaillant avec des données sur la biodiversité. Je montre que le mode de production des logiciels, c'est-à-dire les technologies et les modèles économiques qui les produisent, est biaisé en faveur de systèmes centralisés et uniformisés. Cela m'amène à proposer un programme de recherche interdisciplinaire à long terme pour repenser les outils de développement logiciel afin de créer des infrastructures pour la pluralité. Ces outils peuvent aider de multiples communautés à collaborer sans les forcer à adopter des interfaces ou représentations de données identiques. Le logiciel malléable représente une telle infrastructure, dans laquelle les systèmes interactifs sont des constellations dynamiques d'interfaces, de dispositifs et de programmes construits au moment de leur utilisation. Ma contribution technologique est de recréer des mécanismes de programmation pour concevoir des comportements interactifs. Je généralise les structures de contrôle existantes pour l'interaction en ce que j'appelle des intrications ("entanglements"). J'élabore une structure de contrôle d'ordre supérieur, les intricateurs ("entanglers"), qui produisent ces intrications lorsque des conditions préalables particulières sont remplies. Ces conditions préalables sont appelées co-occurrences. Les intricateurs organisent l'assemblage des interactions dynamiquement en fonction des besoins des composants du système. Je développe ces mécanismes dans Tangler, un prototype d'environnement pour la construction de logiciels interactifs malléables. Je démontre comment Tangler supporte la malléabilité à travers un ensemble de cas d'étude illustrant comment les utilisateurs peuvent modifier les systèmes par eux-mêmes ou avec l'aide d'un programmeur. Cette thèse est un premier pas vers un paradigme de programmation et de conception de logiciels malléables capables de s'adapter à la diversité des usages et des utilisateurs.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Designing and Programming Malleable Software

Keywords : Human-Computer Interaction, Conceptual Modeling, Engineering of Interactive Systems

Abstract : User needs for software features and interfaces are diverse and changing, motivating the goal of making it as easy as possible for users themselves to change software, or to have it changed on their behalf in response to their developing needs. However, in my opinion, current approaches do not address this issue adequately: software engineering promotes flexible code, but in practice this does not help end-users effect change in their software. End-user and live programming systems help users customize their interfaces by accessing and modifying the underlying source code. I take a different approach, seeking to maximize the kinds of modifications that can take place through regular interactions, e.g. direct manipulation of interface elements. I call this approach malleable software. To understand contemporary needs for and barriers to modifying software, I study how it is produced, maintained, adopted, and appropriated in a network of communities working with biodiversity data. I find that the mode of software production, i.e. the technologies and economic relations that produce software, is biased towards centralized, one-size-fits-all systems. This leads me to propose a long-term, interdisciplinary research program in reforming the tools of software development to create infrastructures for plurality. These tools should help multiple communities collaborate without forcing them to consolidate around identical interfaces or data representations. Malleable software is one such infrastructure, in which interactive systems are dynamic constellations of interfaces, devices, and programs assembled at the site of use. My technological contribution is a reconstruction of the programming mechanisms used to create interactive behavior. I generalize existing control structures for interaction as entanglements, and develop a higher-order control structure, entanglers, which produces entanglements when particular pre-conditions, called co-occurrences, are met. Entanglers cause interactions to be assembled dynamically as system components come and go. I develop these mechanisms in Tangler, a prototype environment for building malleable interactive software. I demonstrate how Tangler supports malleability through a set of benchmark cases illustrating how users can modify systems by themselves or with programmer assistance. This thesis is an early step towards a paradigm for programming and designing malleable software that can keep up with human diversity.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Contents

List of Figures

Acknowledgements

An extended network of collaborators, facilitators, mentors, and friends contributed to the production of this thesis and to my growth as a researcher during my doctoral studies. I am grateful for the encouragement, shared wisdom, assistance, critiques, and care of everyone I mention below and more.

This work was done in engagement with three research communities that were my intellectual homes for three years and change: the ex)situ research group at Université Paris-Saclay, the center for Participatory Information Technology (PIT) at Aarhus university, and the Temporary Comp Collective, a loose international network of technologists with shared interests in philosophical and political perspectives on programming and software. These groups formed a nutrient-rich substrate that supported my everyday research activities of reading, writing, thinking, listening, and talking. Each community introduced me to their conceptual tools, shared their histories and open questions, and invited me to build with and on them.

ex)situ was my daily workplace and site of many enriching PhD seminars and coffee break discussions. I am thankful for all my fellow PhD students and interns, who maintain a reciprocal, nurturing work environment, to which I hopefully contributed. Aarhus University was my alma mater, and members of PIT were consistently present on the periphery of my research life through online conversations and occasional visits back and forth between Paris and Aarhus. I especially appreciate Susanne Bødker for inviting me back to Aarhus on multiple occasions, and lending me an office on my visits.

By contrast to these established Human-Computer Interaction research groups, the Temporary Comp Collective is nascent and consists of members in multiple places and disciplines. We are connected by

a desire to articulate how programming and software might be attuned to political values that contrast their historically industrialist ones – such as social equity and citizen participation. This desire was reflected in our online reading groups and at the Salon des Refusés workshops at the <PROGRAMMING> conference, which I had the pleasure to co-organize. In this open-ended space of possibility, Luke Church, Colin Clark, Andrea diSessa, Jonathan Edwards, Joel Jakubovic, Stephen Kell, Clayton Lewis, Mariana Mărășoiu, Tomas Petricek, and Oli Sharpe facilitated early steps in developing my own long-term research interests and extending my disciplinary boundaries.

Among these communities, a few individuals were my sustained partners in research as mentors and collaborators. Michel Beaudouin-Lafon was my thesis supervisor. He was an attentive editor of my work, and helped me improve as a writer, programmer, and presenter. Michel also gave me the independence to pursue my project through its twists and turns. While this rarely led to straightforward outcomes, I am grateful for the opportunity to repeatedly question my own questions. Clemens Klokmose set me on the path toward this work when he showed me a strange piece of web technology hackery called Webstrates during my master's degree studies, leading to an independent study project, a master's thesis, and applying for a PhD position. He was my first researcher role model, and has given much pragmatic advice and reassurance at important moments. I met Antranig Basman a few months into the PhD, and he quickly became my closest collaborator for the rest of it. He helped me ask many surprising, frustrating, and necessary questions, in our long-running instant messaging chats and in his family's home in London. This dissertation would be unrecognizably different without his influence. Among my fellow doctoral students, Carla Griggio and Midas Nouwens shared my interest in characterizing and dismantling the software application. They showed me what that might look like through their research and many stimulating conversations.

I was lucky to carry out studies, program prototypes, and co-author papers with Antranig Basman, Simon Bates, Ghita Jalal, Clemens Klokmose, Wendy Mackay, Nolwenn Maudet, Midas Nouwens, and Andrew Webb.

I am indebted to the participants of the multiple interview studies I helped carry out, who generously shared their time and inspiring insights. In particular I would like to thank Dimitri Brosens, Keith Erickson, Adam Huggins, Andrew Simon, and the iNaturalist team,

whose contributions are presented in this thesis.

Stéphane Conversy, Jean-Daniel Fekete, Myriam Lewkowicz, Nicolai Marquardt, and Jutta Treviranus constituted the jury for my defense. It was a joy to answer their thoughtful questions and critiques, and I carry them with me into my future work.

Wendy Mackay, the director of ex)situ, made (and continues to make) a massive effort to pass on the craft skills of research to all of ex)situ. Gladys Bakayoko, Alexandra Merlin, and Emmanuelle Theneau, administrative workers at Laboratoire de Recherche en Informatique, helped me through many bureaucratic hoops and were patient with my bad French. I had the great pleasure of mentoring Jiali Liu and Miguel Renom during their master's thesis projects. Raphaël James was a great co-instructor in the Fundamentals of Human-Computer Interaction course. Randy Trigg, whose research has inspired me greatly, sent me his thoughts on the relationship between tailorable systems and participatory design.

Moving to and living in Paris was periodiodically overwhelming. The many wonderful friends I made there provided a stimulating life around the work, and helped me survive and thrive. Carla Griggio and Germán Leiva were my adoptive parents for my year and a half in Orsay. Andrew Webb, The English Speaking RPG in Paris community, and the Glassjaw Appreciation Society were my partners in tabletop role-playing adventures. John MacCallum helped transfigure despair into acceptance. Jean-Philippe Rivière and Liz Walton got me in shape. Léna Delval and Stacy Hsueh kickstarted my cultural life in Paris by taking me to many great gigs. Oleksandr Zinenko and Nacho Avellino patiently helped me navigate the bureaucratic entanglements of the university and the state. François Maes and Jeanne Nietsczhke were standout housemates at the Fondation Danoise. Bradley Kennedy sent me poetry and recipes. Jingyi Li shared many good pictures of vape shops. Vanessa Peña Araya was my favorite dance partner.

I gratefully acknowledge the sources of funding that made this research possible. I was funded throughout my PhD by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme grant agreement n° 695464 "ONE - Unified Principle of Interaction", and my work with Nolwenn Maudet and Andrew Webb was partially funded by ERC grant n° 321135 "CREATIV: Creating Co-Adaptive Human-Computer Partnerships".

1

Introduction

We live in an age of ubiquitous computing. In the developed world, it is commonplace to own, use, and have access to multiple digital devices[1] with different formats, running multiple domain-specific applications (apps). These technologies are increasingly essential for everyday life, including participation in political and economic life. Concurrently, people experience diverse and ongoing mismatches between what digital tools offer them and what they need: a word processor may function well for most users, but be cumbersome for those who often need to reference or copy material from outside the editor. An app's shortcuts may help some users become more efficient, but frustrate others who simultaneously use other apps with different shortcuts for similar commands. These kinds of mismatches may be everyday frustrations for the majority of users, but can also effectively exclude users experiencing disabilities. For example, a mouse-based interface may be unusable by people with temporarily or permanently impaired hand dexterity. In addition to such physical and cognitive mismatches, digital tools may also be mismatched with human social and economic needs. For example, freelance workers' labor value is tied up in digital tools that they have invested in owning and learning, and this value may be threatened when that technology is changed, deprecated, or incompatible with the tools preferred by an employer (Nouwens and Klokmose, 2018).

[1] See, e.g., demographic data about technology adoption in Europe (Eurostat, 2016, 2018) and the United States (Anderson, 2015; Anderson and Perrin, 2017).

In practice, many people become adept at marshaling resources to deal with such mismatches, e.g., creating workarounds (Gasser, 1986), adapting technologies themselves or with the help of more capable

peers (Mackay, 1990b), lobbying for different technologies (Bødker et al., 2016a), or creating support systems for learning (Clement, 1993). This "work to make technology work" is inherent in technology use (Suchman, 2002), but in practice, the most severe costs are passed onto communities that fall outside designed-for norms. The contemporary landscape of interactive software operates as a "designer-knows-best" model that gives users very few opportunities to change their software or to make different pieces of software work together as they see fit. Tailorable systems researchers investigate how software systems can be made more amenable to adaptation, i.e. how to design systems that are as easy as possible to adapt at the site of use, whether by its direct users or by supporting developers (MacLean et al., 1990; Schmidt and Bannon, 1992). However, the seminal research in this tradition mostly studied office environments in the early days of personal computing.

In this thesis, I investigate challenges to user appropriation of digital tools in contemporary software use and production practices, and opportunities for reducing the cost and increasing the value of appropriation. This thesis belongs to the field of Human-Computer Interaction (HCI), which broadly aims to understand technology use and propose future technologies. My research theme is how we may rethink the conceptual model and principles of software to better support ongoing design by its direct users and their peers. My research questions are

- How does today's infrastructure for software production and use shape the kinds of design and adaptations that actually happen?
- How does an ecological view of software in use challenge the theory and practice of tailorable systems, which is largely based around a single uniform system in use by an individual or community?
- How can interactive behaviors be turned into units of software that programmers can gradually add, remove, substitute, and amend?
- How can non-programming users be given effective control of such open-ended interactive behaviors?

1.1 Thesis Statement

Today's infrastructure of software development and use manifests itself in network effects that benefit large platforms and in integration

costs that penalize users who appropriate, tailor, and create digital tools. These lead even small non-commercial software developers to practice top-down, assimilative design that pushes diverse user practices to the margins. While user communities are resilient and creative in working around platform limitations, their solutions are often fragile and clunky. This motivates research into *infrastructures for pluralism*, which reduce the cost of collaboration between communities that have made different choices regarding interfaces, functionality, data representations, etc. I introduce *malleable software* as a design vision that reconstructs the goals of tailorable systems with pluralism as a guiding value. Malleable software is made up of interface elements untangled from the closed worlds of apps, which can be developed by different authors and (re-)combined by end users. I contribute toward enabling malleable software by creating a programming model based on *entanglers*, which makes interactions less brittle and more adaptable compared to state-of-the-art programming tools.

1.2 Research Approach

I developed my thesis in the context of the European Research Council Advanced Grant project ONE[2]. ONE aims to respond to the contemporary multiplicity and diversity of digital tools by building a conceptual model that unifies a wide variety of interaction styles and creates more open and flexible interactive environments (Beaudouin-Lafon, 2017). The conceptual model seeks to define interactive systems as combinations of *substrates* to manage digital information and *instruments* to manipulate substrates, both of which can be (re-)combined at will by users.

[2] http://www.erc.one

The work in this thesis began as an investigation of what kinds of technical infrastructures might support interfaces based on substrates and instruments. Eventually, this work grew to encompass social and economic infrastructures as well, as I came to understand that the productive forces of software, e.g. the languages, editors, architectures, and run-time environments are entangled with particular relations of production, e.g. the ways software is packaged and sold, and the way in which agency and labor is distributed among designers and users. I then steered my research to investigate software adaptation sociotechnically, i.e. both in terms of organizational and technological support.

This thesis follows a typical HCI research pattern of problematizing

current technology use and proposing future artifacts that may ameliorate the discovered problems. Whereas I begin by describing problems occurring in a real case, I have not "closed the design loop" by evaluating my design concepts or mechanisms in a real-world setting. There are pragmatic and conceptual barriers to carrying out such an evaluation. For one, malleable software proposes significantly re-engineering many parts of the technology stack that underlies contemporary software, making a realistic implementation far beyond the scope of a PhD thesis. On the other hand, a toy version of a malleable software system would exclude participants from using their regular digital tools, destroying the ecological validity of a field study. Instead, I have aimed to design and evaluate manageable pieces of a feasible future infrastructure[3].

[3] Kari Kuutti has described a related practical and epistemological challenge in HCI: he argues the field is split between "blunt edge" and "leading edge" research. Blunt edge research seeks to understand technology use in the real world, while leading edge research seeks to develop future technologies. Knowledge from one end is not trivially applicable at the other end. In one direction, we can't necessarily derive general design requirements from studying real practices. In the other direction, we can't necessarily predict whether a given technology will be generally useful across use contexts (Kuutti, 2019).

I apply techniques from across engineering and social sciences. I triangulate methods from theoretical, empirical, and design perspectives (Mackay and Fayard, 1997). Triangulation is a research strategy that combines different methodological approaches to help manage trade-offs between generalizability, precision, and existential realism.

Theoretical Work I take three theoretical perspectives on what software is and how it changes:

- *artifact ecologies and infrastructuring*: software is a kind of mediating artifact occurring in ecologies of physical and digital artifacts used by individuals or groups to support a web of related activities. These artifact ecologies change continuously, through open-ended means ranging from custom development to workarounds (Bødker and Klokmose, 2012; Bødker et al., 2016b; Star and Bowker, 2002).

- *programs and authorial networks*: software is source code and runtime representations of programs, which are the targets of authorial gestures, such as (re-)writing code, composing modules, and instantiating classes (Basman et al., 2018a).

- *interfaces and tailoring*: software is an interface made up of objects such as icons, windows, and cursors, and interactions such as pointing, dragging, copying and pasting, etc., and is changed by configuration and customization, carried out with specialized parts of the interface, such as configuration files, scripts, and plug-ins (Mackay, 1990b; MacLean et al., 1990).

I apply these perspectives analytically in chapters 3, 4, and 5 respectively. The phenomena that each of them bring into focus interact.

Applying multiple perspectives helps me address these interactions, as well as arguable blind spots. For example, one interaction is the relationship between software adaptation practices in the wild and the software engineering notion of flexible programs. Based on the findings of our case study and my design goals, I argue in chapter 4 that the flexibility goals of typical software engineering literature are extremely conservative, and ignore the pragmatic reality that most end users, whose needs motivate adaptation, have no effective access to an even notionally adaptable form of their software. This then leads me to argue for more ambitious conceptions and tests of software flexibility.

Empirical Work I co-designed and carried out the case study presented in chapter 3. The primary data collection method was interviews, with additional screenshots, code, documents, and communications collected from online discussion forums, source code repositories, etc. We analyzed the data using meaning condensation (Flick et al., 2007) in a deductive mode, seeded with the conceptual vocabularies of artifact ecologies and infrastructuring. We used artifact ecology mapping as a further analytical aid (Bødker et al., 2017). In chapter 4 and 5, I analyze programming techniques and runtime forms representative of current software development methods in relation to the design goals for malleable software. I argue that these are unsatisfactory both in terms of software engineering and interaction design.

Design Work I co-designed a programming model based on the novel concept of *entanglers*. I developed a prototype implementation of this model with web technologies, TANGLER. I evaluated TANGLER through demonstrations illustrating the kinds of program adaptations it enables, and compared it to related programming literature.

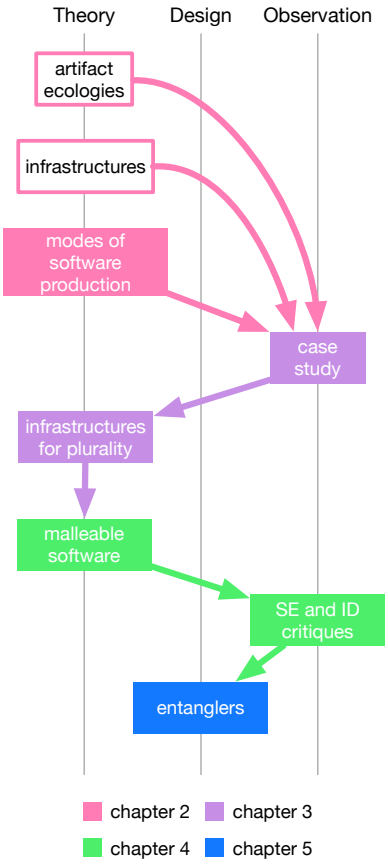

Figure 1.1: Method triangulation in this thesis. 'Observation' is shorthand for empirical investigations. Filled boxes are original research contributions and outlined boxes are existing work that I apply.

1.3 Thesis Overview

I did not carry out the research presented in this thesis in the chapter order. Instead, I have structured the text to present my argument and findings in an orderly way: earlier chapters give a language and context within which to understand later ones.

In chapter 2, I present the theoretical background that grounds the research questions and design approaches of this thesis, reviewing related work on software use, design, and the relationship between the two.

In chapter 3, I investigate the needs for and barriers to software adaptation today in a case study of a biodiversity research network, and introduce the overarching goal of developing socio-technical *infrastructures for plurality*: relations and techniques for producing software that lets diverse user communities collaborate without having to consolidate their technologies.

In chapter 4, I define *malleable software* as one proposed infrastructure of plurality. Malleable software is the concept of interfaces that consist of interactive artifacts that may be freely pulled apart and (re-)combined as part of use, by contrast to the app ecosystems that define software today, in which each app defines an independent interface that may only interact with the outside world through limited, prescribed means such as exchanging files. I describe research problems for implementing malleable software within the disciplines of software engineering and interaction design.

In chapter 5, I describe the design and prototype of a programming model motivated by malleable software. The goal of this model to enable developers to create interfaces and interactions that are robust and flexible enough to function in a software environment where parts of the interface come and go continuously. It does so by introducing the mechanism of *entanglers*, which define interactive relationships as a combination of a *co-occurrence*, a machine-recognizable description of things that may interact, and an *entanglement*, a relationship to be instantiated among those things when they co-occur.

In chapter 6, I conclude with a summary of my contributions, and a discussion of possible topics for future work.

1.4 Publications and Collaborators

While working on this thesis, I presented some of the content in research publications and public presentations. Rather than being a way of sharing essentially finished work, writing, presenting, and discussing my work has been instrumental in the larger research process. It has helped me connect to additional literature, refine my questions and contributions, and begin collaborations.

This thesis comprises material from the following papers:

[HCI.Tools] Philip Tchernavskij. *Decomposing Interactive Systems*. In the HCI.Tools workshop at CHI '17.

[SDR17] Philip Tchernavskij, Clemens Nylandsted Klokmose, and Michel Beaudouin-Lafon. *What Can Software Learn from Hypermedia?*. In Conference Companion of the 1st International Conference on the Art, Science, and Engineering of Programming (PROGRAMMING '17). https://doi.org/10.1145/3079368.3079408

[SDR18] Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. *An Anatomy of Interaction: Co-ocurrences and Entanglements*. In the Conference Companion of the 2nd International Conference on the Art, Science, and Engineering of Programming (PROGRAMMING '18). https://doi.org/10.1145/3191697.3214328

[PPIG18] Antranig Basman, Philip Tchernavskij. *What Lies in the Path of the Revolution*. In Proceedings of the 29th Annual Workshop of the Psychology of Programming Interest Group (PPIG '18).

[InPrep1] Philip Tchernavskij, Antranig Basman, Midas Nouwens, and Michel Beaudouin-Lafon. *Control and Ownership of Artifact Ecologies in a Biodiversity Research Network*. In preparation.

[InPrep2] Philip Tchernavskij, Antranig Basman, Clemens Klokmose, and Michel Beaudouin-Lafon. *Co-Occurrences and Entanglements: A Programming Model for Open-Ended Interactive Systems*. Submitted to the ACM SIGCHI Symposium on Engineering Interactive Computing Systems.

The initial technological framing for my thesis was developed in collaboration with Clemens Klokmose and Michel Beaudouin-Lafon, and presented in [HCI.Tools] and [SDR17].

An earlier version of the case study presented in chapter 3 appears in [PPIG18]. Antranig Basman began the empirical investigation independently and proposed the initial research framing. The full case study appears in [InPrep1]. Antranig Basman and I planned and carried out the interviews and additional data gathering. Midas Nouwens, Antranig Basman, and I contributed equally to the analysis and writing.

Iterations of the programming model presented in chapter 5 appear in

[SDR18] and [InPrep2], and were presented in talks I gave at Aarhus University, OCAD University, the 2019 Common Interactive Object Symposium, and the University of Cambridge. Michel Beaudouin-Lafon and Clemens Klokmose conceived of the notion of entanglements previously, and I began working on elaborating it into a toolkit for interactive systems in 2016. Simon Bates independently developed the concept of co-occurrences and the first prototype co-occurrence engine in connection with the Global Public Inclusive Infrastructure (GPII) project[4]. In May 2017, Simon Bates, Antranig Basman, and I began an ongoing discussion and idea exchange relating to the design and development of highly adaptable software. This led to the creation of the entangler programming model, initially presented along with several early prototypes in [SDR18]. I continued to develop the prototype that turned into TANGLER throughout most of my thesis work, and the versions of the model and prototype presented in this thesis appear in [InPrep2].

[4] https://gpii.net/

2

Background

I review research on three overlapping phenomena of human-computer interaction: the nature of contemporary technology use and production; the relationship between technology design and use; and technological support for continued design and appropriation of technology in use. I also review recent work seeking to characterize how software is produced and used today, and especially how production techniques and practices manifest in use. This literature provides the basic vocabulary of my research, and provides a framework for my research questions.

2.1 Computer Artifacts

To clarify and bound the notion of a digital tool, I take on Bødker's definition of computer-based artifacts. In Bødker's terminology, an artifact is an object that mediates human actions toward other objects or subjects (Bødker, 1991, pp. 36–44). It encompasses the commonsense notion of tools such as pens, but also paper, screens, notations, and graphical user interfaces (GUIs). Artifacts can temporarily "disappear" in use, i.e. letting subjects direct their attention primarily at the object of work. Bødker also defines the concept of a breakdown as a situation in which this mediating relationship is broken and the subject must direct their attention toward the artifact. For example, a pen may cease to put ink on paper, causing the writer to diagnose and fix the issue with the pen, substitute another one, or cease writing altogether. When we analyze a computer interface, we are usually interested in its properties as a mediating artifact: whether it suits the physical and

cognitive capabilities of users, draws attention to the salient objects of work, and supports the instrumental and communicative aspects of routine tasks. Bødker's theoretical framework is cultural-historical activity theory (Bødker, 1991, p. 23), but this definition of artifacts is useful "common coin" for all HCI researchers who want to describe the category of things they study (Nardi, 1996, pp. 10–14). It clarifies that 1) analog and digital artifacts are not different in essence, but mediate human actions differently; 2) what is an artifact is subjective: it is a role assigned to objects in use, not embedded in the process of design; 3) the properties of an artifact are relational: the same app has different properties based on the user, device, activity, and so on; and 4) artifacts can be studied at multiple scales, e.g., we can analyze an app as a whole, but we can also focus on the scrollbar widget as an artifact that reoccurs across apps (sometimes with different properties).

2.2 The Relationship Between Design and Use

In HCI, design is often framed as a formalized process that ends when technology is put into use. In practice, this distinction between design and use is questionable. For one, it implies that once the design process is finished, an artifact should be perfectly fitted to the use situation. This does not square with the actual nature of use, in which there are always more possible contexts:

> Every real world system is an open system: It is impossible, both in practice and in theory, to anticipate and provide for every contingency which might arise in carrying out a series of tasks. No formal description of a system (or plan for its work) can thus be complete. (Gerson and Star, 1986, p. 266)

or as MacLean et al. (1990) put it, *"It is impossible to design systems which are appropriate for all users and all situations."* Furthermore, HCI researchers have extensively shown how technology use is interleaved with design activities. It is precisely because of the many contexts users find themselves in that they often need to adapt or adapt to the capabilities and constraints of technologies: people modify their artifacts, practices, and environments in response to ongoing mismatches and momentary breakdowns among new and old artifacts, tasks, conventions, standards, community members, etc.

The HCI and Computer-Supported Cooperative Work (CSCW) literature has developed several theoretical concepts capturing the everyday work of making technology work, and relating it to more formal-

ized design and development. Articulation work is the labor that precedes and frames primary work activities, i.e. the secondary activities needed to divide, allocate, coordinate, schedule, mesh, and interrelate work activities (Schmidt and Bannon, 1992). Co-adaptation (Mackay, 1990b), informal design (Clement, 1993), design-in-use (Henderson and Kyng, 1995), and intrinsic design (Kaptelinin and Bannon, 2012) all describe how users continually transform their practices and technologies in response to changing requirements and resources. Much of this design work is enacted as articulation work. Tailoring is design work where users modify computer artifacts directly or with the support of technology specialists, e.g. by changing source code, configuration settings, or documents (Trigg et al., 1987). Artful integrations describe the technological artifacts produced by local design processes as hybrid systems combining the artifacts themselves with concrete users, routines, locations, other artifacts, etc. (Suchman, 2002) Infrastructuring describes how these processes cumulatively create socio-technical systems that support work while receding into the background (Star and Bowker, 2002). The activities I have so far referred to as "adaptation" can be variously described through any of these concepts. I apply the theoretical framework of artifact ecologies (Bødker and Klokmose, 2012), which integrates much of this work, and aims to account for use and design specifically in the contemporary context of multiple heterogeneous computer artifacts.

The artist-researcher Olia Lialina echoes these theories in her proposition that users *"formed through three decades of adjusting general purpose technology to their needs"* have become General Purpose Users:

> General Purpose Users can write an article in their e-mail client, layout their business card in Excel and shave in front of a web cam. They can also find a way to publish photos online without flickr, tweet without twitter, like without facebook, make a black frame around pictures without instagram, remove a black frame from an instagram picture and even wake up at 7:00 without a "wake up at 7:00" app. (Lialina, 2012)

2.3 Accounting for Multiplicity with Artifact Ecologies

Computer artifacts are usually used in close proximity and coordination with other physical and digital artifacts. Artifact ecologies are the sets of artifacts that individuals or communities own, have access to, and use. Jung et al. (2008) originally defined the term with a focus on personal devices, and Bødker and colleagues extended it to include hardware and software artifacts at various level of abstraction (Bødker and Klokmose, 2012), and to encompass artifact ecologies developed and used by communities of users (Bødker et al., 2016b). Bødker et al.'s conception of community artifact ecologies has been developed through several studies of a volunteer community distributing locally-sourced organic food (Bødker et al., 2016a,b, 2017). These studies richly illustrate how grassroots communities design their technologies. By contrast to corporate settings, they must rely on whichever resources, knowledge, and labor members can spare. Bødker et al. (2016a) find that the community's artifact ecology develops through a process mixing external circumstance, community strategies, and ev-

eryday tailoring and appropriation tactics. Examples of design stories from the community include the appropriation of one member's old laptop as a payment terminal, a series of discussions and a vote at collective meetings around the adoption of a new payment system, and finding a way to hack a calendar feature into their website after its developer left the community.

Artifact ecologies are a useful analytical lens in my context because they draw attention to the dynamics of adoption and adaptation in today's use contexts of multiple heterogeneous computer artifacts. Community artifact ecologies are seeded with the ecologies of key members and similar communities, as well as available general-purpose tools. They change continually as new tools are introduced, the community and its practice changes, and as members step in and out of deciding roles. As the ecology is renewed, older artifacts tend to linger due to dependencies, familiarity, and the difficulty of integrating disparate tools and information embedded in them. As a result, the community ecology has a complex genealogy and composition, being made up of multiple overlapping ecologies activated in different activities.

I expand on the work of Bødker and colleagues by adopting a critical stance toward the technologies that shape artifact ecology development. The concrete qualities of code and interfaces shape the paths of least resistance followed in artifact ecology development. For example, they co-determine the relative costs of choices such as adopting a general-purpose tool, performing some adaptation on it, or having a specialized tool made. To investigate this shaping process, I apply the ethnographic theory of infrastructures developed by Star and colleagues (Star and Ruhleder, 1996).

2.4 Tracing Extended Socio-Technical Systems with Infrastructures

Infrastructures are the root network of technologies and practices that lie under and support the production, distribution, maintenance, and use of computer-based artifacts. In common use, the word "infrastructure" denotes large, distributed technologies that lie literally and metaphorically beneath more local and tool-like technologies, e.g., the electrical network supports light switches and appliances, road networks and oil extraction, processing, and distribution networks support personal and public transportation, and so on. Star and colleagues have elaborated on this commonsense usage in their development of

infrastructural ethnography (Star and Ruhleder, 1996; Star, 1999; Star and Bowker, 2002).

In their conception, infrastructures are not things so much as socio-technical relations. In other words, infrastructures consist not just of wires and asphalt, but of social relations, conventions and techniques, formal standards and so on. This focus on relations also accounts for the subjective, situated nature of infrastructures, e.g., the electrical wiring in an office may be transparent infrastructure to the worker, until it reveals itself when they spill water in a socket and the whole hallway loses power. These socio-technical relations are themselves recursively infrastructural, e.g., road networks are maintained by people, and other people still manufacture their tools, teach and learn their skills, define and regulate their work, and so on. Star and Bowker (2002) coin the word "infrastructuring" to direct focus to how infrastructures are made up of the ongoing performance of particular activities by many people. They bound and clarify their terminology with eight salient features. Infrastructures:

- are *embedded* in other structures, social arrangements, and technologies;
- are *transparent* to use, invisibly supporting tasks;
- have temporal or spatial *reach or scope* beyond a single event or site of practice;
- are *learned as part of membership*: outsiders and newcomers to a community of practice encounter its infrastructure as objects to be learned about, while members take it for granted (Lave and Wenger, 1991; Star and Ruhleder, 1996);
- are *linked to conventions of practice*: infrastructures shape and are shaped by the practices they support, e.g., the electrical power network is managed according to and affects the daily rhythms of electricity usage;
- *embody standards* that let them plug into tools and other infrastructures, but also lead to conflicts due to different local conventions;
- are *built on an installed base* whose strengths and limitations they inherit, rather than being created anew, e.g. new systems are designed for backwards compatibility; and
- *become visible upon breakdown*, revealing and drawing attention to themselves in power blackouts or bad cellular signals.

These features are somewhat ambiguous in distinguishing infrastructures from artifacts. Indeed, there are significant overlaps in the language and underlying philosophy of infrastructures and artifact ecologies. Both perspectives explicitly include not only how computer artifacts mediate production tasks, but how they support (or necessitate) articulation work. Their units of analysis can be distinguished somewhat by the fact that an infrastructural reading tends to address the extended phenomena that make "ordinary technology use" possible. For example, a keyboard may be more artifact than infrastructure to the individual user when it is clear that the consequences of using one keyboard vs. another are limited to their personal workflows, and that they can easily replace their present one with a new one if they wish. Yet the same keyboard can be read from an infrastructural point of view, e.g., tracing the historical, economic, and technical reasons it has a QWERTY layout.

I apply artifact ecologies to describe the configurations of things that a person or community owns, has access to, and uses, while I apply infrastructures to ask how those things manifest standards, conventions, and similar phenomena.

2.5 Designing Tailorable Software

Software tailoring is adaptation work that modifies computer artifacts directly, e.g. by changing source code, configuration settings, or documents. Tailorable software is designed specifically to facilitate this sort of adaptation work. MacLean et al. (1990) summarize the research agenda of tailorable software[1] when they say that *"it should be as easy to change software as it is to use it"*. Tailorable software research is particularly design-oriented in that it is concerned with describing and designing interfaces, architectures, and organizations that enable end users to configure modify, or integrate their digital tools.

[1] This research program has had various names, including tailorable (MacLean et al., 1990), customizable (Mackay, 1990b), personalizable (Haraty et al., 2017), or adaptable (Fischer, 1993) software. Here I will consistently use "tailorable". It also overlaps significantly with other traditions in HCI, such as end-user development (Paternò and Wulf, 2017) and participatory design, e.g. (Karasti and Syrjänen, 2004).

Tailoring Architectures

Some of this research describes qualities of programs that support different kinds of adaptation work. For example, Trigg et al. (1987) describe four qualities of systems that can be tuned or customized by end-users: *flexible* systems provide generic objects and behaviors that are useful to different users in different situations; *parameterized* systems offer a range of alternative behaviors for users to choose from; *tailorable* systems let users modify behavior, e.g. by building macros,

specializing behaviors, or programming additional behaviors; and *integratable* systems can interface with or be integrated with systems external to themselves.

In addition to work proposing design guidelines for programs, there is a large body of research developing software construction tools and environments that factor in tailoring needs in various ways. These systems range from what are effectively operating systems to more typical programming frameworks. They often define a distinct software development paradigm, such as live object systems in the tradition of Smalltalk (Kay, 1993; Maloney and Smith, 1995; Ingalls et al., 2016), recombinant computing (Edwards et al., 2009), palpable computing (Svensson Fors et al., 2009), and shareable dynamic media (Klokmose et al., 2015). I discuss these design principles and environments in relation to my empirical findings and design goals in chapter 4 and 5.

Tailoring Cultures

Other research focuses on the motivations and means for tailoring in various real-world settings, including the experimental Information Lens email filtering system at a research laboratory (Mackay, 1990a), X Windows during its deployment at MIT (Mackay, 1991), spreadsheet development in various settings (Nardi and Miller, 1991), Computer-Aided Design (CAD) software used by engineers and designers (Gantt and Nardi, 1992), word processors used by administrative workers at a university (Clement, 1993), and labor inspectors in the Danish government (Trigg and Bødker, 1994).

These studies describe tailoring as a social and collaborative web of activities in which people occupy different cultural roles. For example, Mackay describes "translators" as key community members that help curate and distribute customizations, connecting expert developers to new and less technically skilled community members (Mackay, 1990a). Both the organizational and technological makeup of the use setting shapes the emergence of these roles by creating, e.g. opportunities, costs, and incentives. For an organizational example, in Gantt and Nardi's study of CAD users, the workplace paid particular expert users primarily to help other users tailor their systems (Gantt and Nardi, 1992). For a technological example, in Mackay's study of X Windows users, the fact that customizations were created as text files made it easy for most of the community to share and modify them (Mackay, 1991).

Trigg and Bødker (1994) emphasize that tailoring is not only a process of individuating and diversifying software, but also plays a role in collective efforts to standardize workflows, exemplified by how the community they study embeds standard forms and procedures as WordPerfect templates. Clement (1993) illustrates how tailoring is connected with labor relations through two examples of groups of secretaries that have been given computers but few resources to learn how to use them. One group is well-connected enough that they create and share a repertoire of macros and templates, while the other group ends up collectively organizing for their workplace to set aside money and time for training. Clement's study is a concrete example of how tailoring and other means to re-fit practices and technologies are somewhat fungible, i.e. users select design methods that are locally available and effective.

MacLean et al. (1990) present *Buttons*, a system designed to take advantage of both the social and technical tailoring means studied by their contemporaries. *Buttons* is a desktop environment designed for office workers at Xerox EuroPARC, in which the principal interface elements are documents in overlapping windows and buttons. Buttons may be used to execute commands, or gradually unfolded to expose and configure their appearance, command parameters, and the scripts defining their commands. MacLean et al. design the system as a *tailoring architecture* that enables many tailoring techniques ranging from reorganizing and sharing buttons to substituting parameters to modifying scripts or asking more capable peers to do so. They concurrently attempt to create a *tailoring culture* by seeding the environment with buttons encoding some of the participants' common tasks, and by embedding a designer to help users describe their needs and implement tailorings. They report that this combination successfully helped participants obtain a sense of ownership of their tools and to become active tinkerers who can formulate, discuss, and execute small modifications to improve and optimize their workflows.

Tailorable Software Today

While tailorable software research has developed useful design guidelines and principles, the software landscape today is radically different from that of the late 80s and early 90s. The seminal empirical studies of tailoring we reviewed in the previous section were done when computers were primarily desktop machines used in office work, provided and maintained by the workplace. By contrast, there has been limited empirical focus on the technological conditions for adaptation in settings where work is mediated by multiple heterogeneous personal

devices, platforms, and apps.

Haraty et al. (2017) have filled some of this research gap with their recent study of online ecosystems for sharing software customizations. The software use context has moved from work to everyday life and work intermixed, the tailored artifacts have changed from desktop software in the workplace to heterogeneous apps running on personal devices (a game, a text editor, and two productivity applications), and the user/developer communities from co-located professionals to independent users loosely connected by social networks and sharing platforms. These online ecosystems are distributed and networked iterations of tailoring architecture and tailoring culture (MacLean et al., 1990), consisting of different sets of tools, such as repositories for curating popular customizations, and roles, such as "packers" who combine multiple customizations useful for some activity in easy-to-install packages. The productivity applications studied by Haraty et al. are particularly interesting because they enable customizers to create integrations, programs that combine the functionality of multiple apps and services. For example, integrations might be automatic trigger-action rules such as "send me an SMS notification whenever I receive an email from a particular address", or more efficient interfaces, such as a command to search the contents of a website directly from the desktop.

By contrast to earlier tailoring studies, where subjects are defined by shared organizational membership, Haraty et al.'s subjects are defined by shared use of particular customization platforms. This invites us to study tailoring practices in the context of communities bound together by a shared practice rather than a shared platform. These communities encounter the routine incompatibilities among devices, software, and data that are the norm in many artifact ecologies (Bødker et al., 2016b). For example, we might ask whether integration authoring tools such as those studied by Haraty et al. are used to bridge such incompatibilities, or whether incompatibilities devalue customizations because they are coupled with artifacts that are unusable in some contexts. We begin to answer these questions in our case study (chapter 3), although we do not find any use of integration authoring tools. Instead, the integrations found in the case study are either combinations of manual workarounds and appropriated general-purpose tools, or custom software integrations created by professional programmers.

2.6 The App Paradigm

Design-in-use and tailoring activities occur even in the absence of designed tailoring ecosystems. Instead of tailoring architectures, the technical conditions for these activities are defined by the textual and run-time representations of software, as well as the techniques and processes employed in making and using it. They are not only defined by interfaces and app architectures, but also by programming languages, version control systems, file formats, APIs, etc. These components collectively make up the material of software, in that they determine how it may be (re-)shaped. Some recent research helps us describe salient features of this material today.

I name the current paradigm of software construction and use "the app paradigm" after one of its most visible characteristic phenomena. Apps are a way of packaging and distributing software on desktop, mobile, and web platforms. Though the term and many of the key features of app-based software have existed at least since the personal computer revolution of the 1970s, the socio-technical infrastructures around it are continuously evolving. There is no agreed-upon definition of what features make something an app, but rather a set of family resemblances that are recognizable across apps. Hence it is helpful to think of apps as a software genre[2]. In this thesis I will mainly talk about the app paradigm as it has existed in the last decade, i.e. after the appearance and widespread adoption of cloud computing, vendor-controlled app stores, and mobile devices and web browsers as platforms for user software.

[2] Kay (1984, p. 4) describes software genres as emerging from particular computing/programming paradigms, whereas in my reading, social and economic phenomena also contribute to the experience of these genres.

The Mode of Software Production

The app paradigm can be described as an infrastructure or a set of infrastructural tendencies. I also draw from another theoretical concept describing extended socio-technical systems, *modes of production*. Modes of production are a central concept in Marxist theory used to describe why people are organized in particular socio-economic structures and employ particular techniques in order to subsist (Marx, 2005). Marx exemplifies several historical modes of production, such as feudalism and capitalism. A mode of production is defined by a combination of forces of production and relations of production, and how they connect to each other. Forces of production refer to the means of labor, such as tools and techniques, and the human labor power that creates and uses them. Relations of production refer to the way people organize themselves around labor, for example whether

they produce the goods they need to survive themselves or whether they trade their labor for wages to purchase those goods.

I adapt this theoretical language to describe the contemporary mode of *software* production. Rather than describing the structure of an entire society and its historic evolution, I limit my analysis to contemporary software development broadly, and a network of interrelated software producers and users specifically (chapter 3). This theory foregrounds questions about how the production of software is organized, who has control over which resources, how ownership is distributed, and how these elements of a mode help reproduce themselves.

One historic example of how the relations and forces of production renew each other can be found in the invention and popularization of *information hiding* in software engineering[3]. Parnas formulated the idea that minimizing information exchange between program modules supports a harmonious design process in large software development projects, because individual developers or teams can work on modules with minimal information about the details of other modules (Parnas, 1971). This strategy reduces the ongoing development costs of distributing functionality across many modules, thus supporting a division of labor that allows for more complex systems to be developed. Parnas' motivation was to establish a more efficient, centralized management process for large, disconnected teams of programmers working in a top-down industrial design process involving prior specification of desired functionality. Today, information hiding is embedded in programming tools, e.g. method interfaces in object-oriented programming, and taught through best practices, e.g. the law of Demeter, a design guideline stating that program modules should have minimal knowledge of and communication with each other (Lieberherr et al., 1988). Thus a particular mode of production, the top-down distributed team model employed by Parnas, becomes embedded in and is reproduced via the tools and methods it motivates.

[3] This example is adapted from (Clark and Basman, 2017).

Software evolution and maintenance

The field of software evolution studies the life cycle of programs after they are first deployed (Bennett and Rajlich, 2000). The software evolution literature has critically studied the fit between the relations of software production and programming methodology.

Bennett and Rajlich (2000) present a staged model of the software life cycle emphasizing the distinct kinds of programming effort needed in different stages, e.g. software architecture is specified in *initial de-*

velopment, may be significantly revised in the *evolutionary* stage, and eventually stabilizes before the system enters the *servicing* stage, where changes are expected to maintain existing module boundaries and relationships. These stages motivate different programming tools and techniques, are differently suited to outsourcing, and bring up different research problems. Their model is explicitly framed around typical commercial software development as it was 20 years ago, i.e. systems:

- are owned by a development team that decides how it should be evolved and when to shift to the next stage;
- have few dependencies, i.e. most of the code involved is written from scratch; and
- may be deployed in multiple versions at once, which are serviced and phased out in a staggered process.

According to multiple surveys, incorporation of new user requirements and changes in the software environment accounts for a majority of professional programmers' efforts (Lientz and Swanson, 1980; Nosek and Palvia, 1990). Many software engineering techniques, such as design patterns, are motivated by the need to plan for future adaptations to programs. Bennett and Rajlich forcefully state that this labor cannot be avoided solely by techniques that rely on designers to imagine likely future requirements: *"The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even* conceive *of."* (Bennett and Rajlich, 2000, p. 76, emphasis in original) In these cases, the cost of performing additional design on an existing system is generally not proportional to the complexity of the added functionality, but to the complexity of the system.

Kell (2009) picks up this critique in the context of system development that relies significantly on composition of existing systems and code, where *"Software grows as islands of functionality, founded on infrastructure including programming languages, UI toolkits, development "frameworks", extensible applications (browsers like Firefox, editors like Emacs) and so on"* (Kell, 2009, p. 1). He argues that the strategy of information hiding contributes to the unmanageable cost of software evolution in this paradigm. The "islands of functionality" Kell mentions are usually connected to their environments by a programming interface, or API[4]. It is tacit in the use of these mechanisms that all new programs will correctly use existing APIs and that no APIs are changed while other programs depend on them. Both of these assumptions are frequently violated in contemporary software development, leading to

[4] I use "API" to refer to any method protocol exposed by a program module, whether as a method interface in object-oriented programming or as a web service endpoint. This choice of words helps me distinguish these entities from user interfaces.

the ongoing necessity of *integration work* to re-align programs with each other. This work is expensive, because conventional programming paradigms lead to assumptions about external APIs being scattered throughout code modules.

Kell's critique illustrates how the mode of production resists the emergence of alternate production modes that do not follow its paths of least resistance. The social structures of software development have diversified since 1971, creating many situations where a program may end up in use by multiple communities who may want to extend or revise its design in ways unforeseen by the original developers. In this perspective, information hiding forecloses adaptation, because unforeseen and local design demands are likely to cross-cut existing module structure, increasing the cost of modifying the program (Clark and Basman, 2017).

The Agile Turn

Gürses and van Hoboken (2017) characterize the contemporary mode of (commercial) software production in their analysis of "the agile turn" in software development. They highlight three recent shifts that have accelerated software production and transformed business operations:

- Waterfall to agile development: the tools and techniques of software development now allow for many quick iterations and "perpetual beta" development, characterized by phenomena such as continuous deployment and testing of new features.
- Shrink-wrap software to services: on the one hand, much software functionality is now outsourced to distributed microservices, and on the other hand, many tools that used to be released as shrink-wrapped products have transformed into ongoing subscription services, e.g., Microsoft Word has become Office 365 and local music libraries and players have given way to streaming platforms such as Spotify.
- PCs to the Cloud: hardware capacity has moved from users to server farms, and new techniques have been developed for managing these distributed computing resources, e.g. containerization tools such as Docker.

Gürses and van Hoboken focus on how these shifts affect privacy engineering and governance. They argue that the agile turn disincentivizes designing for privacy on several fronts. For example, especially

smaller software developers are likely to compose their products by curating some of the many available backend services for data storage, user authentication, etc. This means that end users are defaulted into a variety of more or less opaque service relationships, including data collection, storage, and advertisement policies. This may adversely affect privacy by enabling service providers to pool user data across many apps, e.g. Google trackers are present on the majority of the most popular websites (Altaweel et al., 2015). In another example, the agile turn has enabled producers to gradually change software functionality without informed consent by end users, further eroding their experience of autonomy over their tools. This phenomenon is already problematic with traditional periodic updates of shrink-wrapped software products, which can at least often be delayed or refused (Vitale et al., 2017).

Gürses and van Hoboken's analysis invites us to ask how the agile turn has affected conditions for tailoring. The methods and technologies it has introduced are said to make software construction more flexible, enabling designers to reuse components and revise design decisions. This is at least rhetorically similar to the goals of tailorable software researchers, yet it is questionable whether the traditional software engineering perspective on flexibility, defined as minimizing the cost of adjusting a system to meet a new specification (Christensen, 2010), is even compatible with flexibility in the hands of end users. Kell (2009) argues that traditional modularization techniques premised on information hiding are based on the false assumption that modules will be designed in order, and Clark and Basman (2017) show several historical examples of systems that have successfully been reused by multiple communities while violating best practices for modularization. I investigate this contradiction in the case study in chapter 3, and in chapter 4 I elaborate a critique of software engineering flexibility based on those findings.

Apps and Precarious Labor

Nouwens and Klokmose (2018) characterize the use side of the app paradigm in their analysis of apps as essential tools of non-standard knowledge workers. They interview writers, designers, and consultants who are non-standard in the sense that they are not consistently attached to one workplace but work with different employers on a per-project basis. Nouwens and Klokmose focus on three salient qualities of app ecologies:

- Expectation of symmetry: apps are typically designed with the as-

sumption that all collaborating users will use the same product or product line;

- Document-app relationships: apps often rely on a proprietary file format that maximizes access to the particular capabilities of the app, but may be less legible by other apps;
- Updates: apps periodically change at the behest of producers, which can cause breakdowns for users as interfaces are rearranged or older features break.

These qualities manifest themselves in patterns of behavior that impose undue costs on non-standard workers. Workers are expected to acquire and learn their own apps, and a large part of their labor value is tied to the skills and data they build up with their digital tools. Concurrently, employers expect workers to be flexible about apps, file formats, and workflows, e.g. adopting local collaboration tools and conventions on a per-project basis. This tension often causes workers to settle on the lowest-common software denominator, i.e. the app or file format that is most broadly legible, at the cost of their ability to consistently use or build up proficiency with apps that help them work most effectively. The workarounds constructed to avoid these compromises impose other costs, such as extra work to manage divergent files and transfer and transform data across apps.

Participants express their preference for some qualities that have emerged with newer, web-based digital tools. Zero-install tools significantly increase the ability of workers to convince collaborators to adopt their preferred tool. Sharing links to documents rather than files removes much of the burden of managing divergent files and file types. Web tools in general free workers from having to anticipate or be aware of hardware and operating system compatibility issues, as the web is increasingly a de facto standard software environment.

Based on their findings, Nouwens and Klokmose argue that HCI researchers have a responsibility to design technologies that do not disrupt the precarious labor value of users. They propose a set of design goals to help mitigate the issues they find. These include supporting technological asymmetries in collaborations, increasing users' control over changes made to their apps, and improved methods to enable transfer of embedded skills and data among apps. These design goals are highly related to the goals of tailoring, as asymmetry is a natural consequence of the ongoing fitting of digital tools to local needs, and as Trigg and Bødker (1994) point out, the same tailoring techniques used to individuate software are used to institute local standards and

conventions that ease collaboration.

2.7 Key Takeaways from Chapter 2

I adopt the view that digital tools are artifacts mediating human activities, occurring in assemblages called artifact ecologies. Assembling and adapting artifact ecologies is part of the ubiquitous and ongoing design work carried out by user communities. The conditions for both formalized design and this design-in-use work are shaped by the infrastructure of software production. Infrastructures are the "root network" of technologies and practices that lie under and support the production, distribution, maintenance, and use of computer-based artifacts. These socio-technical systems can also be understood through the lens of modes of production, a Marxist concept that describes the materials, techniques, labor, and relations involved in producing and making use of goods. In the contemporary software landscape, production is characterized by web and cloud platforms, microservice architectures, and continual testing/development, while use is characterized by routine incompatibilities among devices, software, data, and community practices.

These points invite us to consider several research questions:

- How does today's infrastructure for software production and use shape the kinds of design and adaptations that actually happen?
- How does an ecological view of software in use challenge the theory and practice of tailorable systems, which is largely based around a single uniform system in use by an individual or community?

3

Contemporary Software Design and Use in a Biodiversity Network

To understand the means and ends of technology adaptation today, we carry out a case study of a biodiversity research network. The network consists of professional and citizen researchers, conservationists, and software developers who collect, process, and use biodiversity data. We interview stakeholders across five communities of practice in this network, and trace the artifact ecologies each has constructed to carry out their work. We analyze the different kinds of design interventions carried out by different communities, and how they are supported or resisted by the underlying socio-technical infrastructure. We find a tension between the need for diverse technological supports appropriate to each community of practice, and the need for scale and collaboration among different communities. Based on my findings, I identify research problems for technologies that support more pluralistic design.

In the previous chapter we saw adaptation – in its many forms – as a way for (collectives of) end users to participate in and enact control over the design of computer artifacts. Bødker et al. (2016a) and Nouwens and Klokmose (2018) have documented how people today assemble ecologies of digital tools and the kinds of routine problems they encounter in trying to make use of them. In this chapter, I expand on this work by investigating how artifact ecologies are constructed and adapted beyond the boundaries of individual users or communities. The goal is to illuminate how the app paradigm interacts with the use patterns of adaptation, such as sharing, combining, creating, and

tailoring digital tools and documents.

We[1] present a case study of a network of professional and citizen scientists, conservationists, and software developers engaged in collecting, processing, and sharing biodiversity data. Through interviews with six stakeholders in this network, we trace the ongoing process of creating, sharing, and maintaining their personal and community artifact ecologies. We supplement this core data by collecting conversations from public forums, issue trackers and other web venues attached to these communities.

[1] This case study was carried out with Antranig Basman and Midas Nouwens, and I use "we" when describing our shared work.

The network under investigation includes two volunteer-run projects that collect and curate biodiversity data about their local environments, a non-profit organization that develops a popular tool used by many such projects, a conservation agency, and an international consortium that accumulates biodiversity data from all over the world and makes it available for researchers. These organizations are detailed in table 3.2. By analyzing multiple communities, we can trace how the different productive forces they have access to and their corresponding relations of production have implications for the structure of the resulting artifact ecologies. Our empirical approach lends us a look at the life cycle of digital artifacts, in particular the multiple relations they participate in as objects of design, tailoring, and use within and between communities of practice.

The background for this study is that Antranig Basman was in contact with our primary informant, AS, for about a year prior to formal data collection. AS founded and leads Biodiversity Galiano (GBP), a citizen science project involving his local community in mapping the life on Galiano Island in British Columiba, Canada[2]. The topic of the dialog has been the technological support needs of grassroots biodiversity projects. These conversations were not collected as data, but informed our research questions, interview plans and analysis methods.

[2] We asked the research participants interviewed for this work whether they would like to be credited or anonymized. Those who asked to be credited are identified in table 3.1. They are identified by initials throughout the chapter in line interview data reporting conventions.

Biodiversity data is collected by professional biologists, citizen scientists, and enthusiasts noting the occurrence of a particular life form in a particular location, and may include physical specimens or samples, photographs, drawings, etc. Usually the first data processing step is identifying the life form, i.e. assigning a formal name associated with a recognized taxon. Later on it may be processed to produce lists, maps, reports, and different kinds of visualizations. The processed data is used, for example, by biology researchers to study relationships among species and environments and by conservationists to track the presence of rare, threatened, or endangered species. These activities,

and the surrounding work to facilitate them, are distributed across the network we study.

Individuals and communities in the network create, adopt, share, and adapt computer artifacts to support their activities. This means that the network contains multiple co-existing design practices, e.g. the iNaturalist team develops a set of web and mobile apps for hundreds of thousands of users, while GBP has assembled a hybrid ecology combining off-the-shelf, tailored, and custom tools used by himself and his immediate collaborators. We analyze these different design practices and how they are shaped by the dominant forces and relations of software production.

We are interested in:

- The different needs for stakeholders in this network to design their artifact ecologies;
- The productive forces that they employ in this design work, and the problems they encounter in doing so; and
- How these capabilities and problems reflect the technical tendencies of the contemporary mode of software production.

Our contributions are three-fold:

1. We map how multiple overlapping communities involved in biodiversity research design their artifact ecologies separately and in collaboration;

2. We analyze how the contemporary mode of software production shapes the design interventions that these communities can enact, and how the resulting artifact ecologies fail to fit the needs of small collectives of users; and

3. We critically discuss these conditions, and suggest research directions for software artifacts that enable user collectives to more economically design more useful artifact ecologies.

3.1 Method

Planning our case study was a challenge for a few different reasons. Many field studies in HCI make use of *a priori* boundaries, such as

professional practice, organizational membership, or shared activities, that help researchers select sites and participants. These boundaries also enable researchers to apply heuristics for determining appropriate sample sizes, such as statistical power and saturation. Our case study started with a single viewpoint, that of AS, and a research interest in the extent to which communities such as AS's could enact ownership of their digital tools and data. We uncovered other actors and artifacts involved in the production and use patterns of GBP's artifact ecology through our interviews. The infrastructures that enact the life cycles of these tools are complex, extended phenomena. We cannot (extensively) list all the possible constituents of these infrastructures, let alone gather and analyze data about them. Focusing on one site or community, such as iNaturalist users or members of GBP, would be counterproductive to our research goals, because we risk ignoring the seams, negotiations, and breakdowns between artifact ecologies and communities. At the same time, our goal is not to be complete, but to draw enough of the picture with enough depth that we can see the causes, methods, and consequences of design interventions.

In our case there is *"a myriad of alternative ways of formulating the object of ethnographic study with no assumption about the totality or unity of the object"* (Karasti and Blomberg, 2018). Therefore, we have to adopt tactics for drawing a useful partial picture of our case. For example, we revised our conception of the field under study as we engaged with it (Karasti and Blomberg, 2018). Each interview included questions to probe for other stakeholders and contrasting perspectives. We pursued those connections that we thought were likely to further inform our research questions. Some details of the case that we considered highly salient were unavailable to us, such as details of the reasoning, context, and process of some of the past design decisions described by participants, which they could not recall or were not involved in[3].

[3] By contrast, Bødker et al. (2016a) had access to detailed traces of the negotiations around past design decisions in their study of a volunteer association distributing organic produce through the association's meeting minutes.

Another challenge was that all the researchers were based in Europe, while the majority of participants were in North America. Therefore our study did not involve any on-site data collection such as observation. This practical constraint limits our ability to capture tacit knowledge and day-to-day patterns of activity in the studied communities. We compensate for this limitation to some extent by our use of critical incident interviewing technique, collection of online communications, and our direct involvement in some of the design interventions we recount.

We applied several tactics to bring infrastructural relations into focus. These tactics were adopted from previous literature, e.g. (Bødker et al.,

2016a; Karasti and Blomberg, 2018). They involved looking for:

- Seams: sites of incompatibility between technologies or communities;
- Breakdowns: situations where infrastructure becomes visible because it fails;
- Design interventions: actions taken to reconfigure artifacts in response to changing use situations, including workarounds, programming, adoption of new artifacts, and so on;
- Artifact biographies: the phases of an artifact's development, distribution, and maintenance, and the associated sites and actors (Pollock and Williams, 2010); and
- Data traces: data that capture some of the details of breakdowns, design interventions, artifact development, etc.

Participants

We interviewed six stakeholders from five organizations (Table 3.1). We recruited participants via snowball sampling, i.e. asking each participant about other, similar and contrasting communities and stakeholders they interacted with. In looking for participants, we focused on potential "infrastructural allies", participants who need to understand infrastructures as part of their daily work (Beaulieu, 2010). This led us to select participants who were actively involved in facilitating the work of collecting, processing, or using biodiversity data, e.g. by selecting, sharing, tailoring, and developing digital tools for themselves and their communities. We stopped recruiting when we felt that we had drawn up a substantially wide-ranging field to show several contrasting contexts for design work. This evaluation was also informed by external time constraints encouraging us to continue to the next phases of the study. Recruiting and interviewing happened over three months.

Table 3.1: Overview of our participants.

Participant	Occupation	Interview (h:mm)
Andrew Simon (AS)	Founder and main facilitator of GBP	1:25, 1:26
Kem Luther (KL)	Co-founder and database maintainer at MBP	1:12
Adam Huggins (AH)	Restoration Coordinator at GCA	1:57
Keith Erickson (KE)	Executive Director at GCA	1:05
Dimitri Brosens (DB)	GBIF Open Science Ambassador	0:53
P6	Funding and collaboration outreach at iNaturalist	1:13

The network includes several distinct communities of practice (Lave and Wenger, 1991, p. 98–100), including professional and academic biologists, amateur naturalists, land managers and restorers, and software developers. These communities are connected by collaboration, data exchange, and software production and use.

The five organizations are briefly described in table 3.2. They are related as follows: GBP and MBP are geographically adjacent projects with similar goals, and are in the process of pursuing direct collaboration. GCA shares a location with GBP and has exchanged data with them. The iNaturalist organization develops the iNaturalist platform for collecting and identifying biodiversity data, which is heavily used within GBP. GBIF receives data from many partner organizations, including iNaturalist, which it makes separately accessible through its own infrastructure.

Organization	Location	Purpose	Members
iNaturalist[a]	Distributed internationally	Develops and maintains the iNaturalist platform	9 staff members, >750,000 users across ~31,000 projects
Global Biodiversity Information Facility[b] (GBIF)	Distributed internationally	Maintains an open access database of biodiversity data and infrastructure for exchanging biodiversity data	Has sub-organizations in 58 countries and more inside other organizations, 38 open data ambassadors
Biodiversity Galiano[c] (GBP)	Galiano Island, BC, Canada	Volunteer-based group that collects biodiversity data through iNaturalist, organizes community events for data collection, education, etc.	1 main facilitator, handful of key organizers, ~100 contributors of observation data over all time, ~500 Facebook members
Metchosin Biodiversity Project[d] (MBP)	Metchosin, BC, Canada	Volunteer-based group that facilitates professional biology surveys and organizes educational events for the public	5 organizers/facilitators
Galiano Conservancy Association[e] (GCA)	Galiano Island, BC, Canada	Surveys, manages, and restores land to protect biodiversity	8 staff members, seasonal student interns

Table 3.2: Overview of key organizations.
a: `inaturalist.org`
b: `gbif.org`
c: `biogaliano.org`
d: `metchosinbiodiversity.com`
e: `galianoconservancy.ca`

Data collection

The interviews were semi-structured and the questions were open-ended. We used Mackay's adaptation (2002) of Flanagan's critical incident technique (1954), where questions are intended to draw out specific instances, problems, or highlights of their technology use. We asked participants to walk through recent or memorable examples of their work, to explain which tools and collaborators they worked with, what specific problems arose in their work, how they changed procedures and technologies in response, etc. This technique encourages interviewees to give detailed descriptions of specific situations, rather than abstract statements about their general experience.

We interviewed each participant once, except for AS, whom we interviewed a second time to follow up on recent developments at GBP. The interviews were conducted via video/audio conferencing (Skype[4]). Each interview was recorded and lasted between 53 and 116 minutes. We transcribed the recordings, including pauses and non-verbal interjections (laughter and exclamations), while omitting minor hesitations and false starts.

[4] skype.com/en

We also collected data in the form of artifacts shared with us by participants. These included web pages, reports, an instructional video presenting AS's biodiversity data curation workflow, and the history of exchanges between AS and the iNaturalist community on the organization's Google Group. Finally, we read public web pages run by the key organizations (table 3.2), including iNaturalist's documentation, user forums, and source code repository on GitHub[5], as well as tutorials and documentation from GBIF. This data supplemented interviews with examples of the output and everyday usage of the involved platforms.

[5] A very widely used tool for developers to share and manage code. github.com/inaturalist/inaturalist

Analysis

We used meaning condensation (Flick et al., 2007) to summarize data content and interrogate it in the context of our research questions. First, three investigators separately performed inductive, open coding on overlapping subsets of the interview transcripts, where they labeled all parts of the transcript that were relevant to our research questions with a short summary code. Then all investigators discussed the open codes and decided which topics were most salient to our study. In this discussion we reframed codes in terms of artifact ecologies and infrastructures, and defined categories to group codes, including descriptive categories such as *stakeholders*, *activities*, and *artifacts*, and more interpretive categories such as *artifact ecology negotiation*. One investigator then applied these categories to re-code the complete interview transcripts and tied together the resultant codes in descriptive statements. Finally, we used the additional data collected from public forums to detail or revise the generated descriptions where necessary.

Additionally, we created artifact ecology maps representing the stakeholders, activities, and artifacts of GBP and MBP (table 3.2). These maps summarize the constellations of technological artifacts that our participants' communities use to carry out their work. This method was adapted from the mapping exercise used by Bødker et al. (2017). We used these maps to structure our descriptions of the design practices of these communities.

We focus on four non-exclusive categories of software artifacts, distinguished by the kinds of objects they mediate relationships to:

- *Data artifacts* are the materials that participants collect, generate, and process, e.g. spreadsheets, maps, and other documents.
- *Function artifacts* are the interfaces and units of functionality that mediate work, e.g. apps, widgets, and APIs.
- *Expression artifacts* are the source code that produces software, as well as secondary artifacts related to code, such as incremental versions, executing virtual machines, and plugins.
- *Installation artifacts* are the actual machines and networks on which software is executed, and the software that mediates access to them, such as cloud services for renting virtual machines[6].

[6] These could also be called infrastructural artifacts, but we use "installations" to avoid confusion with the theoretical term.

Since our interviews focused on the concrete experiences of participants, our descriptions of community artifact ecologies are more detailed around their immediate activities, and less detailed at relatively peripheral ones. We describe artifacts at different levels of abstraction based on their role in participants' use and design activities. For example, AS approaches Facebook as a uniform entity, whereas in his use and appropriation of iNaturalist, different parts of the web platform, such as project pages and embeddable widgets, mediate different activities. While the ecologies we map include devices such as cameras and smartphones, as well as other physical artifacts such as field journals, we do not pay particular attention to these kinds of artifacts in our analysis.

3.2 Communities and their artifact ecologies

Each of the organizations in our case study has different motivations, challenges, and resources that shape their relationships with technology and design practices. I report our findings on these aspects and how they have shaped the development of each organization's artifact ecology.

Biodiversity data and taxonomy

The practice of assigning and maintaining taxonomic classifications is central to some of the recurring issues in the network. Taxa, i.e. the distinct categories of species and evolutionary relationships among them,

are continuously revised as part of the practice of biology. For example, genetic analysis of specimens may be used as a justification to split or merge previously held taxa. When a biologist identifies an observed organism with a certain species concept, that identification is tied to an authoritative source. Taxonomic authorities are usually curated databases, which compile concepts developed in peer-reviewed research articles. Different fields of evolutionary biology often agree on a standard reference, such as the World Register of Marine Species[7]. However, it is up to the professional judgment of individual biologists whether they agree with a reference on a case-by-case basis. For example, they may prefer to subscribe to an authoritative reference that they perceive as having high standards for inclusion, or disagree with one that they consider to lag behind the newest research. As a result, the taxonomic categories in biodiversity databases are frequently the sites of contention.

[7] marinespecies.org

iNaturalist

iNaturalist is a mobile and web platform whose principal functions are to let users collect biodiversity data by photographing plants and animals in the wild to submit the geo-referenced and timestamped pictures to a public database, and to identify each others' observations by assigning formal species concepts to them. iNaturalist users may submit and identify observations freely, but often do so as part of projects, which are user-created pages dedicated to some particular purpose, such as collecting observations of seabirds in a particular location. At the time of writing the app has over 750,000 users.

The platform started life as a student project at UC Berkeley, and is now supported and developed by a formal organization, also named iNaturalist, financed by the California Academy of Sciences, National Geographic Society, and additional grants. The iNaturalist organization has nine staff members, five of whom are developers whose efforts are distributed among the web, iOS, and Android apps, a child-oriented species identification app called Seek, and management of development and operations. P6 is primarily responsible for communication with collaborating organizations and funding sources. One example of a collaborating organization is the City Nature Challenge[8], a yearly competitive bioblitz where cities across the world compete to collect and identify the most observations over a few days. In its latest iteration, the event spanned 159 cities.

[8] citynaturechallenge.org

The iNaturalist apps are developed according to contemporary best practices of software engineering, including the use of agile methods

and cloud services described by Gürses and van Hoboken (2017), e.g. storing observations photos in Amazon S3[9], using the Google Maps API[10] to display and interact with maps, and Microsoft's Azure platform[11] for operating the testing and deployment pipeline. In addition to the main website, the organization has agreements with multiple organizations that deploy localized instances of the iNaturalist frontend, including local control of presentation, language, outreach, and data sharing, but connected to the centralized backend.

[9] aws.amazon.com/s3

[10] cloud.google.com/maps-platform

[11] azure.microsoft.com

As iNaturalist has evolved, it has had various encounters with competing and cooperating platforms. These encounters are indicative of how large software projects respond to diverse user needs. For example, previous to her employment at iNaturalist, P6 was commissioned to build a "portal" that reused iNaturalist's existing backend, but offered some variant features and presentation, including using the branding of a different institution. She found that the only available choices were either to reimplement the function from scratch, or to fall back to a lightly restyled version of iNaturalist's frontend interface. From her perspective, this resulted in a lot of wasted effort and an inferior product:

> [...] where we landed, was that the project essentially ran out of funding before we ever even fully duplicated the core function of iNaturalist, and to give you an example, you could create your iNaturalist account, add observations, you couldn't add more than one photo to an observation, you couldn't rotate your photos, and you couldn't edit your observations once you uploaded them [...] We spent a lot of money, and we ended up with a totally inferior product, and I really wasn't interested in trying to do that again. So I've definitely steered people away from wanting to do something similar [...]

In another case, a localized variant of iNaturalist, Natusfera, was forked[12] from it due to different legal requirements for data storage. As the respective sources have ceased to be compatible, the forked project has become unable to take advantage of updates that have happened to the original platform. This is likely to eventually cause Natusfera to become economically unsustainable.

[12] In software development, forking is the act of copying a project's source code and starting independent development on it.

P6 reports that in at least three cases users of smaller, competing platforms have made agreements with iNaturalist to migrate their observation data to the bigger platform. In these cases, the iNaturalist team has built custom importer tools that interface with the source platform's API to copy its observation data and accounts. These platforms were then subsumed and became disused.

iNaturalist's developers are engaged in ongoing design negotiations with its user communities in public and private forums, e.g. its now-retired Google Group, new community forum, or GitHub page.

One prominent topic of discussion is whether and how the platform should mediate taxonomic disputes, i.e. conversations about whether a given taxon is valid or validly assigned. iNaturalist users who frequently contribute species identifications may be promoted to the status of curators, giving them the ability to introduce and update taxa. In particular, curators are expected to respond to flags, a moderation request that any user can assign to taxa, observations, and comments on the platform. This forms a second level of collective data maintenance: all users collectively assign taxa to observations, whereas curators determine whether taxa themselves are legitimate. As iNaturalist has grown, this flag-and-moderate system has come under strain, as the provenance and rationale behind a particular taxon may be difficult to recover. This is an issue in particular because some taxonomic niches are not well-covered by any standard reference, such as butterflies or fungi, creating the potential for frequent disputes. Recently, the developers have introduced a more detailed tool, Taxon Frameworks, that is intended to help *"reining in taxonomic free-for-all"*[13]. Taxon Frameworks are a structured format for relating taxa in iNaturalist's database with outside references. For example, the framework for the phylum Chordata is sourced to the World Register of Marine Species, but includes 15 deviations, such as shifting the class Mammalia from the subphylum Tetrapoda to Vertebrata.

[13] inaturalist.org/blog/19977-introducing-taxon-frameworks

iNaturalist receives requests from user communities to add features related to their specific niche practice. For example, P6 reports that one recurring request is for the platform to enable reports that a particular species is absent in an area. Representing non-presence is necessary for rigorous biodiversity sampling and monitoring work. iNaturalist considers this outside their current design scope:

> We have some kinda hacky workaround-y things, but nothing like the sort of full-featured monitoring tool that they would like. And I think that it's allowed us to focus on our strength and continue to improve that, without getting pulled in too many directions by trying to be a tool that can meet everyone's needs in the context of biodiversity science.

Instead, iNaturalist attempts to facilitate these communities in integrating the platform with their local tools, e.g. through their public API, or by agreeing to periodically send them exported subsets of iNaturalist's observation data. The integration difficulties experienced by GBP indicate that this kind of integration work would be costly.

Global Biodiversity Information Facility (GBIF)

GBIF is an international organization whose purpose is to run an open-access online repository of biodiversity data. It is made up of a consortium of partner organizations, including national and academic institutions. In some cases, these partnerships are directly between GBIF and an organization that has the resources to periodically publish data to GBIF's repository, such as iNaturalist. In other cases, GBIF partners with institutions whose employees take on the role Open Science Ambassadors (OSAs), who are responsible for mobilizing data from local projects into GBIF's infrastructure. DB is one such OSA, working at the Research Institute for Nature and Forest in Flanders, Belgium[14]. GBIF also funds the training workshops, conference attendance, and periodic international coordination meetings for OSAs.

[14] inbo.be/en

GBIF has developed its own infrastructure for managing, providing, and entering data, including the Integrated Publishing Toolkit web app for submitting data sets. They also rely on specialized tools and standards, such as the Darwin Core XML file standard for biodiversity data[15].

[15] dwc.tdwg.org

DB's primary task is to oversee the transfer of data from academic and citizen science biodiversity projects in his region into the GBIF repository. This involves finding partner projects, establishing relationships with them, and either meeting to assist them in uploading their data through GBIF's Integrated Publishing Toolkit (IPT) web app, or receiving an exported file and uploading the file himself. Data sets from local projects have to be processed to fit GBIF's standards and to be understandable and usable by researchers accessing it in the future. IPT provides limited support for this processing work – such as renaming columns in a data set to ones that are part of Darwin Core – but in practice it requires an open-ended set of skills and technologies. According to DB, this is both because IPT is due for a redesign, and because *"every data set is a different story"*.

Therefore, DB and his community create and share scripts, written e.g. in R[16], tools such as the open-source data cleaning app OpenRefine[17], and tutorials on how to use these. Scripts and tutorials are created and shared locally, in the research institute where DB works, and made available through their GitHub account. Sometimes, these tools are shared in the scientific community of Biodiversity Informatics to which DB belongs. For example, the yearly conference of their interest group TDWG, presents an award for scripts and tools that are useful to the general community.

[16] r-project.org

[17] openrefine.org

These specialist tools let DB create custom automatic routines for partner projects that he receives periodic data from: *"We create a script or something which is built on their data model, so if we run the script again I get the data [into the GBIF format]."* Notably, outside of outright erroneous data collection or formatting, DB does not tell partner projects to adopt GBIF's standards, such as the Darwin Core (DwC), in their own databases, as he argues they know better what is useful to them: *"I say to people, do not use DwC to push in your data. So if you have a research project, look at the DwC and if there are any terms which you can use in your data sets, well, that's good, use them, but don't try to push everything in DwC [into your data schema] because it can really disturb your research projects."*

Biodiversity Galiano (GBP)

GBP is a volunteer-run citizen science project whose goal is to document and raise awareness of the flora and fauna of Location 1. The community's work involves collection of biodiversity data through the iNaturalist platform, organizing community events such as bioblitzes, – intensive one-day biological surveys – nature drawing workshops, and kayaking trips.

The project was started by AS, who is a graduate student in biology. One of the rationales of GBP is that participants can contribute to the knowledge of local biodiversity by confirming historical records. Around the same time that AS started GBP, he began a related project of collecting and organizing all of the historical data from Location 1 about plants in his field of expertise. This historical data was sourced from private lists prepared by professional biologists in AS's network, a database at a provincial university, or through contact with curators at a provincial natural history museum. Another is that Location 1 is classified as a rare biogeoclimatic zone, having the highest density of species at risk in its province. Concurrently, a large majority of the land is private property, which is effectively inaccessible to independent researchers. Thus citizen scientists are also generating new records of species not previously known to occur in Location 1. One of AS's key activities is curating a database in which he collates data gathered by project members with historical data.

Members of the GBP community vary in their level of engagement, from largely passive Facebook group members, to frequent or one-off observation contributors who use iNaturalist to submit geo-referenced photographs of species encountered in Location 1, to members who maintain the community by helping organize events, sharing pho-

tographs and observations on Facebook, etc. AS and these community champions have made many efforts to engage the community, including publishing biodiversity scavenger hunt checklists in the local newspaper, making a sign on the local harbor inviting visitors to contribute to iNaturalist, etc.

GBP's technological needs are as non-uniform as its membership. On the one hand the broader community of contributors and organizers requires accessible and usable tools that present minimal adoption costs. On the other hand AS needs specialized tools for curating, analyzing, and representing data. For example, his data curation workflow cannot happen inside iNaturalist. Firstly, it involves several kinds of data that do not fit in iNaturalist's ontology, such as historical observations collected by AS, taxonomic identifications unrecognized by iNaturalist, and annotations indicating whether a species has been confirmed by a GBP contributor yet. Second, the iNaturalist interface is specialized for entering and viewing data, rather than manipulating it, i.e. observations can only be accessed and edited one at time. Thirdly, iNaturalist manages an internal list of "sensitive species", for which observation coordinates are automatically obscured, so AS would also run the risk of destroying parts of his historical data by uploading it to their database. This is not so much an issue because AS cannot carry out his curation work with other tools, but rather because this aspect of his work does not benefit from iNaturalist's high usability and accessibility by the wider GBP community:

> You know it's providing a benefit for the user, it's a great tool for learning, at a very basic level for people, but beyond that for people to more meaningfully engage in sort of an analysis of the data [...] I'm just interested in making this data more accessible to people who are starting out from a more novice standpoint, but who could go places in learning about the species diversity of the region, and all these various ecological patterns that I find fascinating.

AS has periodically interacted with the iNaturalist organization through its Google group, reporting bugs, participating in discussions, and requesting new features. However, he has found that the tendency of the developers is to refine iNaturalist as it exists, rather than widening its aims to meet GBP's specific needs. We reviewed these public threads (n=42) and found that threads which raised an issue or posed a question were responded to with, in order of commonality, no response (14), suggested workarounds within existing functionality (8), explanations of existing functionality (4), explanation that a requested feature was out of scope for the project (2), and explanations that a future feature might account for the raised issue (2). These responses

came both from developers and other members of the user community. Furthermore, AS feels that if the project would live inside iNaturalist, it would lead to a lacking sense of ownership and transparency for GBP:

> [W]hen you pour so much of your life into something... say I have a bunch of business cards printed out [laughs] it's a very non-lucrative business this business of documenting biodiversity, but when I meet somebody and I pass out this business card, currently all I have is the iNaturalist link on it. But that's a link to a website created by, well, it's iNaturalist, it's not my website. So partly I've been motivated to take ownership of this project as a curator and do something that's a little bit more original and more authentic to the purpose.

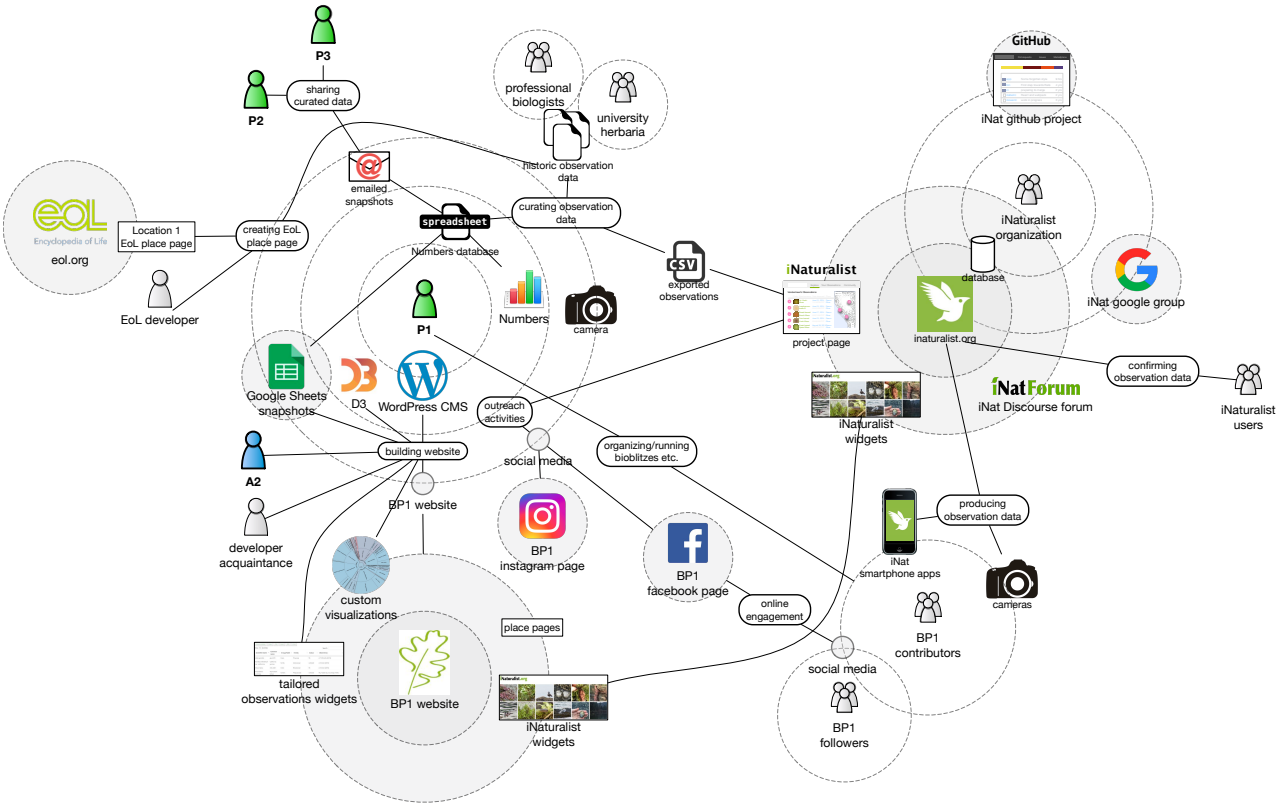

Figure 3.1: A map of the personal and community artifact ecology of AS and GBP. Personal and communal artifacts are grouped in transparent circles around individuals and groups. Online platforms and artifacts are grouped in grey circles around their domains. Text in ovals are activities, connected to participating people and artifacts.

In response to this situation, AS has gradually substituted and augmented iNaturalist with additional artifacts that he has more control over (figure 3.1). Currently, he primarily uses the MacOS-specific spreadsheet app Numbers[18] for data curation, and has created a community website with the WordPress Content Management System (CMS)[19]. He has developed a "precise algorithm" for data curation, which involves periodically querying iNaturalist for recent observations in Location 1, extracting a CSV file, reducing the incoming data to the

[18] `apple.com/numbers`
[19] `wordpress.com`

first observation of each distinct species, integrating those data with his master spreadsheet, and cross-verifying entries with the historical record and taxonomic authorities. In the process of building the website, he has learned to create page templates with WordPress and data visualizations with D3[20]. This process has been partially self-taught, using online tutorials and tailoring example artifacts, and partially done with the help of developers AS knows socially, including one of my co-investigators. The broader GBP community is also involved in making the website, e.g., through photography excursions to create content for the website's "place pages", which present the ecosystems of particular locales within Location 1. At the same time, iNaturalist remains an integral part of the community artifact ecology because it is unparalleled as a data collection tool for the broader community of novice naturalists.

[20] A JavaScript library for programming data visualizations. `d3js.org`

Operating and maintaining this artifact ecology requires ongoing work and various glue technologies. For example, the process of updating the observations on a *place page* involves exporting observation data from iNaturalist, transforming it to fit AS's master database format, and then copying the results to a Google Sheets spreadsheet containing observations for the particular location, which a WordPress widget on the GBP website automatically queries to update the appropriate observations widget. The resulting integration is also unsatisfactory in how it factors publicly accessible and private artifacts. With iNaturalist and the community website, it is easy for GBP members to contribute data and to see the project's outputs, but much harder to participate in all the work between these two:

> So iNaturalist has done great things, but at the same time it's also tied in in a similar way to the pipelining of the data in one direction, producing a byproduct, all these benefits for the individuals who are out there exploring and stuff, but limiting to a certain extent the way in which those users can make use of these data. (AS)

AS's long-term goals for GBP is to make the organization self-sustaining, and to support a network of similar organizations in the region by sharing the tools and practices GBP has developed. To that end, he collaborates with surrounding projects such as MBP, and has taken a job with a conservancy in the province in which his task is to recreate the data curation and presentation tools of GBP in their location.

Metchosin Biodiversity Project (MBP)

MBP is a volunteer-run project whose goal is to document and raise awareness of the flora and fauna of Location 2. Like GBP, it facilitates

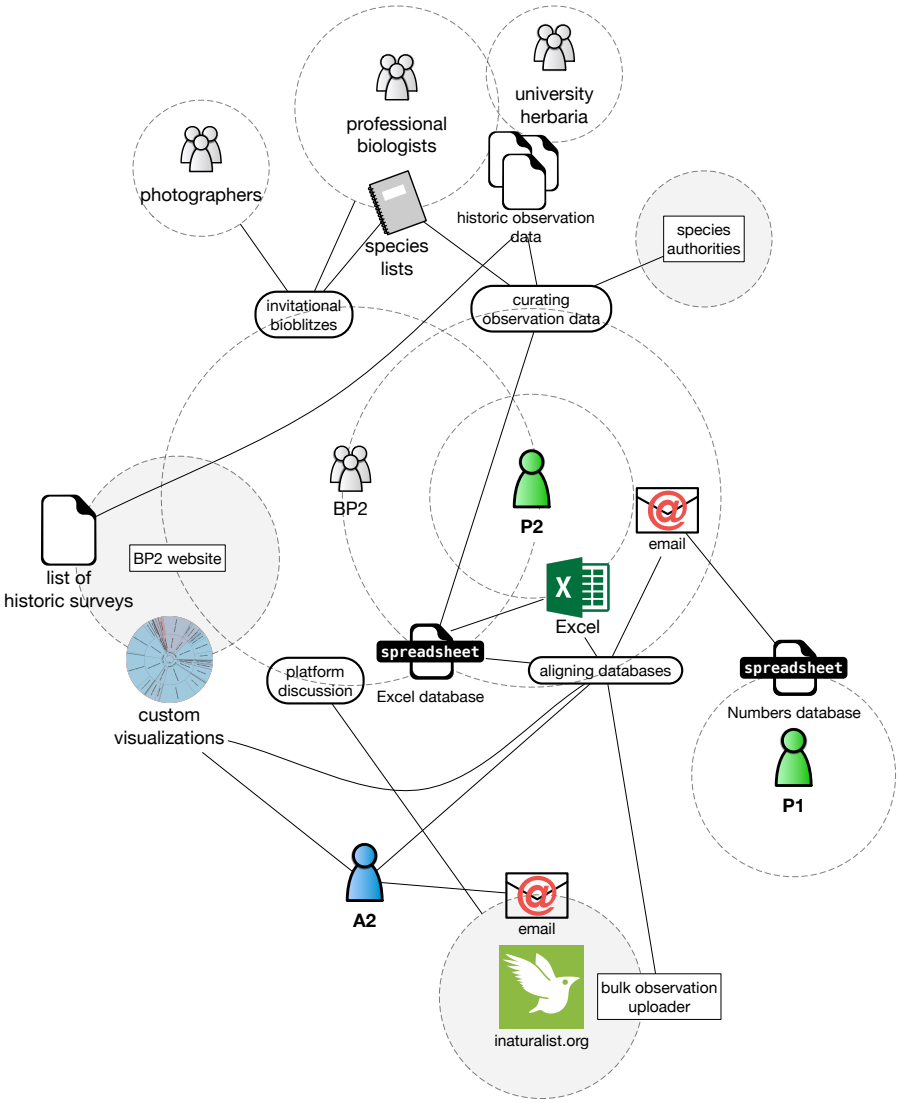

Figure 3.2: A map of the personal and community artifact ecology of KL and MBP.

the collection of biodiversity data and organizes educational events for locals. However, MBP is different in a few respects. It is run by a stable group of five volunteers, rather than one founder and community members who volunteer on a case-by-case basis. It facilitates data collection by inviting professional biologists – including experts in species that require specialized skills to detect and identify, such as algae – to bioblitzes twice a year, rather than relying on the general public and iNaturalist. Separately from these surveys, it organizes educational events for the public. Location 2 is a rural district with some large areas of undeveloped land. MBP has focused their surveys on this land to support future conservation efforts: *"[W]e have this side goal, not only enumerating species in Location 2, [but also] providing some basis, some database for future transitions of this land."* (KL)

KL has a background in Computer Science, and acts as MBP's local technology expert. In particular, he created and manages their database, which is a collection of Excel[21] spreadsheets into which he periodically enters the species lists generated at the MBP bioblitzes, as well as data from historical biodiversity inventories of Location 2. He also manages the community's website, which presents community events and resources.

[21] office.microsoft.com/excel

By contrast to GBP, MBP defers to the data collection and representation practices of invited experts. Rather than encourage usage of a uniform tool such as iNaturalist, KL collects data gathered by experts in whichever form they prefer, digital or analog (figure 3.2). Relatedly, MBP has less need to foster continuous engagement with the project by amateurs, and therefore does not operate e.g. a Facebook page. Instead, their community outreach consists more in organizing periodic events and producing educational resources. As a side-effect of these factors and the fact that we relied on KL's perspective of the community, our map of MBP's artifact ecology is less dense than that of GBP.

In addition to GBP and MBP, the larger region contains two further biodiversity documentation projects focusing on their respective local environments. Members of these communities know each other through the social network of naturalists in the region, and sometimes exchange expertise and resources, e.g. participating in each others' bioblitzes and sharing their data organization schemes. AS is interested in building these occasional meetings into a regional network of biodiversity projects that can more closely share and develop data and tools, and support larger-scale and comparative analyses of biodiversity across the region. As a move in this direction, AS has shared with KL the code for a custom taxonomic data visualization developed by

one of my co-investigators.

Like AS's master spreadsheet, KL's spreadsheet for collecting MBP's survey data is adapted to locally developed conventions for representing observations, i.e. it has a different set of columns with different names. To make the visualization developed for AS compatible with KL's data, their spreadsheets must be aligned. KL first attempted to transform MBP observations to the GBP format via Excel scripting, but this only worked for observations that could be detected as appearing on both data sets. Subsequently, KL realized that MBP's data set could be semi-automatically extended with the necessary metadata by bulk uploading his observations to iNaturalist's database via their CSV import interface. Over the course of assembling his database, AS sourced his version of this information from this table by hand, cutting and pasting from iNaturalist's UI for his observations. This information is also publicly available via iNaturalist's JSON-formatted API feeds, but neither AS nor KL had access to sufficient development resources to make use of these. In practice even the clever workaround discovered by KL involved significant articulation work. The CSV format accepted by iNaturalist upload is a fixed format, different from both the re-download format and also from KL's original database — so this integration involved two stages of conversion and re-export from KL's spreadsheet tool, ending up with some extra columns which could then be incorporated in yet another conversion to produce the rendition of the data suitably similar to AS's data that it could be ingested by the visualization tool.

The cumbersome nature of this articulation work would become prohibitive if scaled up from this small local interaction between three participants (AS, KL, and iNaturalist), to the wider network of cooperating regional naturalists envisaged by AS. Even without the use case of shared visualizations, both AS and KL would have an incentive to align their data with iNaturalist's schemas since this would gain access to its built-in map-based and list-based visualizations as well as its community. But without the ability to adopt iNaturalist's own persistence as the authoritative data store for either project, this alignment requires this cumbersome and unscalable pipeline to be operated on a regular basis rather than just as a one-off task.

Galiano Conservancy Association (GCA)

GCA is a conservancy whose goals are to preserve, protect, and enhance the quality of the human and natural environment in Location 1. The typical work of a conservancy involves surveying and acquiring

land, negotiating covenants on privately held land that legally limit its future development, and monitoring and restoring biodiversity on land they hold. GCA also does less typical conservation projects, such as applying sustainable agriculture principles to grow and sell produce in a forest, or collecting and reseeding plants that were historically cultivated by First Nations peoples[22] in the area to turn another forest into a foraging source and site for social and educational engagement with and for indigenous people. Biodiversity data is primarily significant to GCA because it is used to prioritize which properties to acquire, protect, and conserve. Documenting the presence of recognized at-risk species or ecosystems can also provide social and legal leverage for GCA's work.

[22] indigenousfoundations.arts.ubc.ca/terminology

AH is a restoration coordinator and KE is the executive director of the agency. Both of them have a wide range of responsibilities, including defining and managing projects, applying for funding, as well as being *"boots on the ground"* that e.g. map properties, survey ecosystems, and collect seeds for propagation.

GCA's common technological resources are several computers, some specialized devices such as GPS units and cameras, a file-sharing server, and a website. They do not have any dedicated technological support staff or developers, and thus their artifact ecology is defined largely by the off-the-shelf technologies its members bring with them. In addition to its 8 regular staff members, GCA relies on the labor of seasonal student interns, who might e.g. work on defining a monitoring plan for a property.

GCA keeps track of its projects through spreadsheets, maps, reports, etc., which have accrued in their local file-sharing server for over a decade. These data are inter-related in multiple dimensions, e.g. through references and version histories. The organization as a whole has not converged on settled schemes for naming, structuring, and versioning files and folders. This situation makes it difficult to keep track of what data GCA has in their possession, and sometimes causes serious breakdowns. In one case, KE discovered that a previous student intern had accidentally corrupted the trail map of one of GCA's properties, meaning that the whole mapping effort has to eventually be redone. Some of this work happens occasionally, e.g. when an intern performing a round of measurements for a long-term monitoring project also recovered and reorganized all the previous measurements for that project. Due to their lack of personnel or financial resources, neither AH nor KE think it is likely that GCA will be able to design, maintain, and enforce data storage policies.

AH has convinced GCA staff to partially ameliorate this situation by migrating GCA's data from the file-sharing server to Google Drive. His hope is that this will reduce the labor of accessing and sharing data, substituting the work of installing and operating remote desktop software to access the file server with sharing URLs. Google Drive also has built-in file versioning, though it does not keep versions older than 30 days for files not created with Google's apps, e.g. PDFs and maps. By contrast, in the above case of data loss, the problem was only discovered a year after the trail map file was corrupted. Furthermore, Google Drive offers little help with the problem of creating and enforcing schemes for naming and structuring the data, and even less help for the problem of tracking related data. So far, GCA has only partially migrated their data and workflows to this new setup.

3.3 Contrasting Design Practices

I now discuss how the artifact ecologies and design practices in our case study manifest tendencies of the contemporary mode of software production. By comparing the design work of different participants, we will see how the kind of labor they do, the amount of it, and the realistic outcomes are shaped by the productive forces they have access to and the relations of production they operate under. I argue that software production tools embed a centralizing tendency, which negatively affect the ability of grassroots communities to derive value from their design efforts.

Relations of production in the case study

The network we study is made up of several overlapping and interacting relations of production. We see three broad types of relations, characterized by the role of designers relative to the use situation. *Platform owners* design tools for an open-ended group of users, embodying the user/designer distinction of typical commercial software production. *User-designers* design tools for their or their community's practice, occupying the spectrum of tailors and related roles documented in tailorable systems research. *Mediators* design in the space between communities, enabling them to cooperate.

The first two categories are congruent with the practices of extrinsic and intrinsic design described by Kaptelinin and Bannon (2012). The third category is somewhat orthogonal to this distinction. I emphasize it because designers enabling cooperating among communities and

their artifact ecologies played an important role in our findings.

Platform owners develop and maintain custom software. They require specialized skills and tools to organize distributed development, test and deploy systems, track bugs, etc. This relation is characterized by a large number of users relative to designers. Consequently, designers working in this role must evaluate which aspects of users' practices to focus on and support. For example, we saw iNaturalist in use as part of personal education, academic projects, environmental restoration work, biodiversity monitoring, etc. While the platform owners are aware of these diverse use contexts, they must focus their limited resources, e.g. by defining iNaturalist's core functionality as *"helping people record biodiversity occurrence data [...] and crowdsource species identifications associated with those records"* (P6).

Communities of practice typically invest in off-the-shelf digital tools that partially fulfill their needs as starting points for their artifact ecologies, e.g. GBP started out with an iNaturalist project and Facebook group as key artifacts. Off-the-shelf tools are attractive both because they may offer appropriate interfaces and because of the network values that come with high adoption rate, e.g. presence of valuable data, supportive community, stable funding and maintenance. We saw this rationale in effect in the migration of user communities and their data from smaller iNaturalist competitors. Network values also explains why some communities lobby platform owners to expand platform functionality for their particular needs, e.g. the iNaturalist users who requested biodiversity monitoring features. It is likely that these users do not have the resources to develop or maintain a high-quality monitoring tool of their own.

User-designers help fill the gaps between what these tools provide and what a community needs, resulting in hybrid ecologies such as GBP's (figure 3.1). They assemble these systems by *"happenstance, community strategies, and everyday tailoring and appropriation tactics"* (Bødker et al., 2016a). A key value for user-designers is the feasibility and cost of opting in and out of aspects of platforms they design around. For example, GBP's website is a more desirable public representation of their work compared to that initially offered by iNaturalist's project pages, but requires AS to manually recreate many of the data flows that are automatically handled by iNaturalist's backend.

These relations of production overlap because the artifacts they create and adapt are held in common between communities due to collaborations (e.g. GBP and MBP) and producer/user relationships (e.g. GBP

and iNaturalist). This sometimes results in tensions, when desired or actual design interventions made by one community affect other communities. The different design stories illustrate different ways of dealing with this tension. For example, AS was unable to effect change in iNaturalist to make it support GBP's goals beyond data collection, so he resolved tension by working around it, gradually complementing iNaturalist with additional artifacts.

Mediators design at the seams between communities to avoid or resolve design tensions and incompatibilities. DB tries to minimize tension by encouraging data sharing partners to maintain their local data schemas rather than adopt the standards used in the GBIF repository. AS and KL created a workaround pipeline to align the observation data from their respective projects for visualization. In both cases, the intervention helps the respective communities accomplish some level of interoperation between their ecologies, but at the cost of repeated work for the mediator. In DB's case, he is paid for this work, and has a suite of specialized tools for efficiently re-processing data. In AS and KL's case, it is unclear if they will repeat this process in the future or invent a less clunky workflow.

Productive forces in the case study

In addition to the different relations of production, we found that our participants made use of different tools and techniques to perform design interventions. For example, when new collection data is uploaded to iNaturalist, the data are automatically made accessible and visible to a combination of programs performing database requests and processing the result into rendered HTML. By contrast, to make the same data present on the GBP website, AS has to manually operate a multi-step process involving exporting CSV files, reformatting them for his master database, and copying that data to an online spreadsheet connected to his website via a table widget he found online and adapted (figure 3.1).

These productive forces occupy a spectrum between use-like and development-like. These poles indicate different trade-offs between required skills, expressive power, and down-the-line maintenance costs. Broadly, use-like tools and techniques require less specialized technical skills to deploy, but result in less sophisticated designs compared to development-like ones. Table 3.3 gives an overview of some key design tools employed by our participants.

	Data	Function	Expression	Installations
Design-in-use	email, spreadsheets (offline & offline), maps, text documents, CSV files	iNaturalist, Facebook & other apps, WordPress	templates & scripts, APIs	shared file server
Software development	databases		large programs, frameworks, version control systems, test suites	cloud services

Table 3.3: An overview of the kinds of artifacts that make up the artifact ecologies we surveyed, roughly grouped by adjacency to the practices of user-designers and large-scale software development.

The hybrid ecologies produced by user-designers are useful, even essential, to their communities, but they have limited avenues for improving their usability. For example, it is cheaper for AS and KL to exchange data by emailing each other CSV files than to program and run a bespoke interface, but their solution has limited power in that they have to manually do the work of extracting and sending data, as well as the more taxing work of translating from one party's data schema to the other's. A capable programmer could likely automate some of this work, which is similar to DB's routine data processing tasks.

The productive forces of software development give platforms owners several benefits in terms of scaling up systems to handle many users and large bodies of data. For one, they can create effective and usable interfaces by re-organizing interfaces to fit routine tasks and automating the background involved in those tasks. Additionally, development and operations tools, many of them cloud services such as Microsoft Azure and Amazon S3, help them manage the processes of creating and testing new functionality, deploying it, and keeping their platform running and responsive.

However, this ability to scale up also comes with an implication of consolidating around shared design choices. At every level, adopting large-scale software development tools implies community consolidation around certain design choices. Adopting a database requires agreeing on how data are named and structured. Adopting a development ecosystem requires agreeing on one body of source code and therefore on functionality. In iNaturalist's case, there is some flexibility in this regard due to the split between frontend and backend, e.g. localized versions of the website can be translated. The development stories reported by P6 illustrate that the ability of the platform to support divergent functionality is limited[23].

This form of scaling is not compatible with the kind of design that our user-designers and mediators pursued. The user-designers and

[23] There are some antecedents for large software projects that are carefully evolved while factoring in the diverse needs of different communities, but they involve substantial management and community governance work, as in e.g. development of the Linux kernel.

mediators in our case study sought to create tools that bring value to one or multiple communities of practice while letting individuals keep their local data and interfaces as they desired. Technological support for design-in-use in an era where digital artifacts are held in common is not simply a matter of allowing for software modification, it must also support maintaining and negotiating differences among communities and individuals.

In summary, the tools of software development support the development of usable and scalable tools, but at the cost of having to manage functionality and data in a centralized way. Gürses and van Hoboken (2017) argue that software increasingly defaults users into service relationships that are non-trivial to understand and resist. I argue that software analogously defaults users into functional relationships that they may not wish to be subject to in a uniform way. In the next section, I discuss what this implies for technological research.

3.4 Infrastructures for Plurality

Based on our case study, I have argued that contemporary tools for software development make it too costly for communities to support their local practices. This cost is too high in several respects. The cost of entry, in the form of the skills and time necessary to make the kinds of digital tools we have been making variants of for decades. The cost of independence is more subtle, in the form of the ongoing work that is necessary to maintain local interfaces, data schema, installations, etc. while still being able to integrate with other tool ecosystems. Both of these costs contribute to a centralizing tendency, a technological and economic rationale for one-size-fits-all design.

I argue that our field should seek to develop productive forces for designing and modifying software that reduce and mitigate integration costs and network effects. I call these potential artifacts *infrastructures for plurality*. A plurality is a social structure that encompasses multiple cultures without giving any of them primacy, but acknowledges *"multiple, located, partial perspectives"* (Suchman, 2002). Infrastructures for plurality are design artifacts that help users cope with the existence of multiple partially compatible systems.

For example, AS and KL's spreadsheet databases are partially compatible in the sense that they theoretically contain data about the same kinds of things. However, they disagree on how spreadsheet columns

are named, which columns exist, and even which data is valid, e.g. where AS and KL disagree on a species authority. As a result of this partial compatibility, AS and KL must go through a lot of cumbersome labor to interoperate their databases, e.g. to compare their data. However, AS and KL are not interested in making their systems fully compatible by consolidating on one standard data representation, because their local choices in this regard support their communities' practices. In this case, a pluralistic tool would enable them to share data with less effort *without* requiring them to change their local data formats, e.g. by automatically transforming data in AS's format to KL's format, and vice versa.

Using this design goal as a guideline, I briefly survey some existing green shoots of software infrastructure that may lead to better design conditions for small collectives of users designing in networked ecologies, and point to some of the gaps between what exists in research and what could have a practical impact in the context of our case study.

Pluralistic data

Tools for managing pluralistic data should minimize the repeated work involved in making use of data across different practices and artifact ecologies. The most notable development on this front is arguably the translation of established app genres – such as spreadsheets and word processors – into the cloud. Collaborative tools such as Google Docs[24] have made several aspects of sharing common document types simpler, e.g. substituting file transfers for shared URLs. As seen at GCA, these tools are desirable to grassroots communities, but little else has emerged to assist small-scale organizations in storing, version-managing and curating their data online. For example, shared spreadsheets do little to assist GBP and MBP in their data alignment task, nor have these tools been shown to support communities in managing data with many versions and inter-relations, as is true of GCA's documents.

[24] docs.google.com

One interesting direction in this area is the Frictionless Data[25] initiative. It defines a portable standard for *data packages* that combines files in various popular formats with metadata that facilitates safe importation into a different environments and repositories. As a backend, this would be appropriate for some purposes of small-scale organizations, such as automating the pipeline from AS's data curation workflow to visualizing subsets of the updated database on GBP's website. However, several gaps remain. Firstly, the currently proposed metadata format for data packages does not allow the expression of relations

[25] okfnlabs.org/projects/frictionless-data

with other data sets. This makes it of limited value in representing distributed data, and in packaging data that may need to be structured differently in different use contexts. Secondly, the initiative has not described user-level tools for creating, manipulating, and transmitting data packages.

To build on this work, I propose developing data bridging mechanisms that establish persistent data sharing relationships between communities without requiring them to agree on policies for naming, structuring, and curating data. The goal is to eliminate or reduce the manual work of moving and transforming data between locations. For example, AS should be able to establish bridges that manage the relationships among his local spreadsheet database, GBP's website, iNaturalist, and MBP's database. Computer scientists have developed a formalism that notionally matches this concept, bidirectional lenses (Bx, 2018). A lens establishes a bidirectional transformation between two data items and maintains them in a consistent state as changes are made to one or the other. However, lenses have yet to manifest themselves in end-user-oriented interface prototypes There is research to be done in evaluating whether current bidirectional lenses are applicable to the kinds of data transformations exemplified in our case study, as well as in creating usable interfaces for establishing and manipulating them.

In discussion with AS, we noted that it is not necessarily desirable for such lens interfaces to be completely automated. For example, they should let users on each end step through updates and dynamically amend the lens with new rules, e.g. rejecting or automatically transforming observations of a taxon considered invalid at one end. Basman (2019) has followed up on our case study with an in-depth analysis of this kind of inter-community data alignment task and the requirements for data formats and user interfaces to support them.

Pluralistic installations

Tools for pluralistic installations should enable communities of users to acquire the hardware support necessary to operate personal and community tools that can be made publicly accessible on the web. This includes the entire chain of support necessary to make a piece of software available in a particular time and place, from the hardware (virtualized or otherwise), the energy and network costs of running it, its connections to sources of data in its community and those of others, the act of installing it, paying any license costs, and ensuring that it stays running stably and free of security vulnerabilities.

Access to these kinds of resources is one of the factors that drives users to ecosystems such as Google's, in exchange for entering service relationships that are sometimes undesirable, such as having one's data tracked and sold to advertisers. Tools in this vein are necessary for hypothetical desirable use patterns such as allowing a community of users to trivially re-deploy the iNaturalist platform with alternate policies, e.g. with respect to data privacy or licensing. This form of dispute lies behind the split between iNaturalist and the localized Natusfera, which had to separate from the main platform due to new legal requirements for data storage at its location.

At least for web applications, this is an area in which there has been significant recent process. The last ten years have seen a proliferation of "Something as a Service" platforms by cloud computing vendors, where "Something" can variously be replaced by "Platform", "Infrastructure", "Software" or others. Each of these provide models where one may rent standardized parts of the infrastructure needed to host a web application, delegating more or less of the responsibility for it to the provider as needed. These rely on various pieces of orchestration technology, many of which have increasingly solid open source implementation, such as OpenStack[26] and Kubernetes[27], which automate the processes of spinning up one or more machines of a particular configuration that cooperate to provide a particular function.

[26] openstack.org

[27] kubernetes.io

This is a highly promising development, although the ecosystem is still far from the state where an ordinary citizen (or expert) could turn up and at the push of a button set up a fully configured incarnation of their own version of a complex platform such as iNaturalist. Two examples of initiatives aiming to help novice developers design and host apps at minimal cost are hood.ie[28] and the Small Technology Foundation[29]. Both provide novice-oriented documentation, curated open-source tools, and novel tools to bridge the gaps between them. They each focus on providing particular useful properties for apps as defaults, such as working equally well on- and offline, or encrypting network traffic. These projects may in time reduce the barrier to communities such as GBP to develop homegrown tools in the same way that user-friendly Content Management Systems such as WordPress have enabled communities to host static websites. However, the resulting tools may be no easier to integrate and combine with each other than those found in our case study.

[28] hood.ie

[29] small-tech.org

Pluralistic function

Tools for pluralistic function should enable end users to keep, reuse, and integrate (parts of) interfaces they find useful in different contexts. These tools should aid users such as AS in assembling hybrid systems without substantial loss of usability or insurmountable maintenance costs. The *Buttons* environment reviewed in the previous chapter (MacLean et al., 1990) exemplifies the kinds of use patterns implied by this goal: users extending their interfaces piecemeal by adding new buttons, sharing them with each other by email, and adapting buttons to new uses by modifying their exposed parameters. Pluralistic interfaces encompass tailorable software, but bring new design challenges stemming from the multiplicity of systems and users. Extending the example of *Buttons*, it should be feasible for users to transport their buttons between different environments with various levels of incompatibility, either by themselves or with the help of a developer. Pluralistic function also requires tailoring techniques to integrate disparate system elements originating in different communities. Ultimately, these requirements imply that it should be possible to disassemble running interfaces into their constituent parts and reassemble them to suit different purposes. Because of this conception of software as a material to be re-shaped in the hands of users, I call this design vision *malleable software*. These goals imply radically different forms for programs and running software compared to those found in the app paradigm. I pursue this design direction in the remainder of this thesis, and review existing work and key research problems in the next chapter.

3.5 Conclusion

In our study of a biodiversity research network, we observed moves to expand and refine artifact ecologies occurring from multiple directions. A given community's artifact ecology is not only theirs, but is also held in common with other communities due to collaborations and producer/user relationships. Therefore, changes to community ecologies were initiated by the community itself, in concert with its collaborators, or occurred in a top-down manner, e.g. when developers abandoned, deprecated, or modified software.

These multiple origins of design were sometimes in tension, when desired or actual design interventions made by one community affect other communities. Sometimes tensions were resolved through collaborative design and sometimes through workarounds, substitutions,

and additions around the changing artifact.

The communities in our case study used a number of techniques to perform design interventions, from adopting an app to sharing a synchronized spreadsheet to programming new apps. Some of these artifacts cost less to deploy in terms of skills, time, and front-loaded maintenance, but are also limited in terms of their design reach, leaving user-designers with some intractable design problems. As a result, the artifact ecologies users create for themselves necessitate ongoing articulation work, whereas developer interventions can take advantage of tools that enable them to automate routine work and change interfaces rather than adding new ones.

Additionally, the available design artifacts have bad properties when designing for multiple heterogeneous communities: in regards to data, function, and installations, it is generally expensive or impossible for two or more communities to share some, but not all design decisions. iNaturalist's relationship to competing platforms and diverging projects is indicative of how one-size-fits-all software design is embedded in the technology stack. Under the current mode of production, control over widely deployed and widely adopted software artifacts needs to be centralized. As a result, platform owners cannot economically do their work without treating users as a largely uniform community, while grassroots communities hoping to collaborate must either perform repeated integration work or consolidate their artifact ecologies.

It should be emphasized here that I do not critique the design practices or choices of iNaturalist or other communities in the case study. Rather, I argue that iNaturalist exemplifies a double bind of software produced under the app paradigm: on the one hand, the platform is extremely valuable to communities such as GBP as a free, highly usable and accessible tool for collecting biodiversity data. On the other hand, those same communities face imposing costs in adapting such systems to work in the context of their local ecologies.

Thus there is an HCI research agenda in reforming the tools of software production to support small-scale, ad hoc, diverse, collaborating communities in making and integrating software at costs they can afford. Based on the design problems encountered in our case study, we propose data sharing tools based on bidirectional lenses, infrastructure-as-a-service tools for end-users, and malleable interfaces as representative artifacts of this reformed technology stack.

3.6 Key Takeaways from Chapter 3

The design practices we trace in the case study support our view that software production tends towards centralized, top-down design. The underlying mode of software production, manifested in social relations and technical artifacts, inhibits user communities from maintaining software that supports local differences in data organization, functionality, and infrastructure. For example platform owners consolidate their user communities under one set of design choices, while grassroots communities invest ongoing labor into bridging between technologies and communities. We argue that the software production stack contributes to this state of affairs. The tools and techniques of software engineering tend to make further design interventions to distributed software uneconomical or impossible unless they fall within a prespecified design space. This infrastructural tendency manifests itself in network effects and integrations costs. These lead even small non-commercial software developers to practice top-down, assimilative design that pushes diverse user practices to the margins. While user communities are resilient and creative in working around platform limitations, their solutions are often fragile and clunky. This motivates research into and design of *infrastructures for plurality*. I envision these as tools and techniques for software design that enable communities to adapt their technologies at a cost they can afford, and especially to reduce the cost for communities to cooperate without having to consolidate their technologies.

4

Malleable Software

I define the design goals of malleable software, which are the focus of the rest of this thesis. Malleable software takes up the ideal of tailorable systems research, i.e. that changing software should be as easy as using it. It refines and refocuses the aims of that research by taking into consideration that people own and use multiple software artifacts in changing ecologies, and that each distinct software artifact is connected to a network of developers and collaborators. Therefore I focus on technological interventions that reduce the reliance on stable, central systems as the targets of adaptations. Instead, malleable software should dissolve the boundaries of discrete systems such as apps, letting users and developers pull them apart and (re-)combine them. I describe the goals and challenges of creating malleable software from the perspectives of software engineering and interaction design.

In the previous chapter we saw two distinct approaches to designing ecologies of digital tools: software development resulting in a multi-app platform, and a mixed approach combining off-the-shelf apps and various glue technologies resulting in a community artifact ecology. I found that each of these approaches offered unsatisfactory trade-offs for grassroots communities of practice: the former approach can result in highly usable and efficient tools, but at the high cost of software development and maintenance. The latter approach is accessible to resource-strapped communities, but at the cost of significant limitations to usability and efficiency. Concurrently, both approaches have trouble accounting for pluralism, i.e. the presence of and interactions among multiple communities with multiple practices and preferences for their digital tools. In this chapter, I describe *malleable software*, a vision for developing and tailoring software that seeks to combine the

benefits of making design interventions accessible and making the resulting artifacts usable.

Malleable software is inspired by the everyday adaptation practices reviewed in chapter 2 and documented in chapter 3. People bring together artifacts they find useful, organize them to fit routine tasks, create reusable artifacts that reify local conventions, etc. Just as people prepare their physical spaces and tools to suit their practices,[1] they prepare digital artifact ecologies by, e.g.,

[1] see e.g. (Nielsen and Bødker, 2004) and (Maudet et al., 2017) for examples from the domains of tax assessment and graphic design.

- installing sets of useful apps;
- organizing them in available screenspace;
- creating template files that can be copied or extended;
- naming and structuring files and folders; and
- using available configuration options.

These adaptation behaviors are limited by the rigidity of software. In particular, they are subject to the boundaries imposed by apps. In general, apps couple what users can do, e.g., changing color, with what they can do it to, e.g., text. Some apps supports collaborating on a particular task, but limit the available tools and working materials. Some apps can be extended with new tools, but are highly specialized, e.g., for illustration or programming. Apps can only be combined in limited ways, e.g., it is possible to work on an image in two graphics apps in turn, but impossible to compose those two apps to make their respective tools available at the same time on the same image.

Malleable software aims to increase the power of existing adaptation behaviors by allowing users to pull apart and re-combine their interfaces at the granularity of individual UI elements, such as toolbars, widgets, menus, documents, and devices[2]. In other words, the goal is to erase the boundaries between apps and create an end-user accessible "physics of interfaces" that dictate how different interfaces and documents can be assembled. Malleable software should enable end users to reuse their favorite digital tools in different digital environments, combine the behaviors of multiple interfaces created by different developers, and recruit developers to modify or substitute interface elements.

[2] I mainly describe malleability in relation to the most common types of user software, in which "UI elements" are graphical. It is not, however, inherently connected to visual interfaces.

Figure 4.1 illustrates the kind of adaptation malleable software should enable. Here, a user of a communication app wants to create and share

data visualizations directly in a message thread. Therefore, she drags some tools into the chat window, including a color picker from her map-making environment and a graphing tool from her spreadsheet environment. She then proceeds to use these in the same way she would in their original environments, i.e. selecting data to visualize, clicking an appropriate graph type, adjusting its look with the color picker, etc.

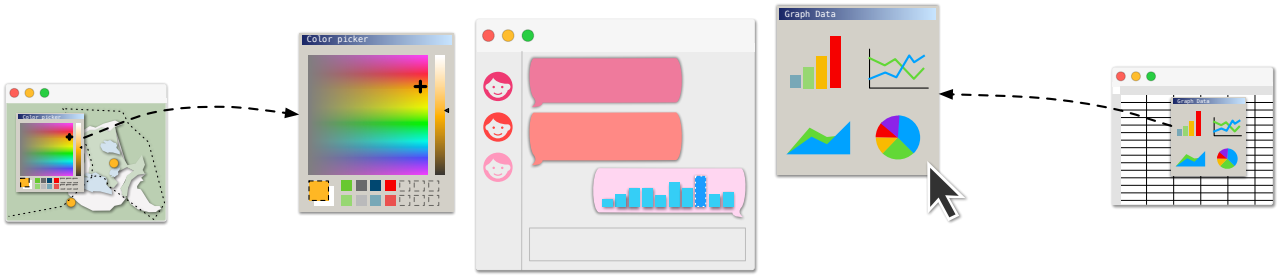

Figure 4.1: Adding visualization tools to a chat environment by extracting and assembling existing interface elements

Malleable software entails a literal interpretation of the qualities of flexible and reusable programs. The former traditionally means minimizing the cost of adjusting a system to meet a new specification (Christensen, 2010), whereas the latter means that a program module can be written once and be executed in several different contexts (Biddle and Tempero, 1998). Malleability means that these properties should manifest themselves in the hands of users. The malleable reuse scenarios I am addressing are significantly more ambitious than those addressed by typical programming techniques, such as design patterns. For example, we will see that the simple-to-describe act of moving a color picking tool from one window to another recapitulates significant programming effort (figure 4.3).

This design vision is strongly influenced by several threads of work in HCI and programming language research. Beaudouin-Lafon (2000) introduced the interaction model of instrumental interaction, in which interfaces are made up of hand tool-like instruments that mediate actions directed toward domain objects. Instrumental interaction extends and refines the design principles of direct manipulation (Shneiderman, 1997) by introducing the mediating artifacts of instruments, which are decoupled from the objects of work. Klokmose and colleagues have subsequently appropriated Beaudouin-Lafon's interaction model for the design of multi-device interactive systems (Klokmose and Beaudouin-Lafon, 2009; Klokmose and Zander, 2010; Gjerlufsen et al., 2011; Klokmose et al., 2015). Beaudouin-Lafon (2017) synthesized this work in his proposal for an interaction model based on *information substrates* that represent digital information in multiple

layers of abstraction, and instruments to manipulate substrates. This proposal was the outset for my thesis work.

Basman (2016) critiques the lacking materiality of software in the sense that programs expose no consistent affordances for modification and maintenance. He argues that software as a whole is unacceptably brittle and rigid, particularly in how it responds to ongoing adaptations. Clark and Shahi (2018) reflect on how the methods and tools of software production might be reconstructed to support ongoing change in use. They propose *material software* that *"provides the power to be adapted, configured, re-presented, augmented, or separated in various ways, without needing to have been part of the original software development process or to have access to 'elite-level' programming knowledge or tooling."* (Clark and Shahi, 2018, p. 5).

I contribute to this work by describing and addressing some of the concrete design problems that come up in bringing this vision into reality. I start with two hypothetical scenarios of software modification that illustrate the chasms that malleable software must cross.

4.1 Scenarios

I describe two hypothetical adaptation scenarios as benchmarks for malleable software. By comparing idealized malleable approaches to the realistic processes for completing these adaptations, I uncover multiple research problems.

Small-Scale Scenario: Moving a Color Picker

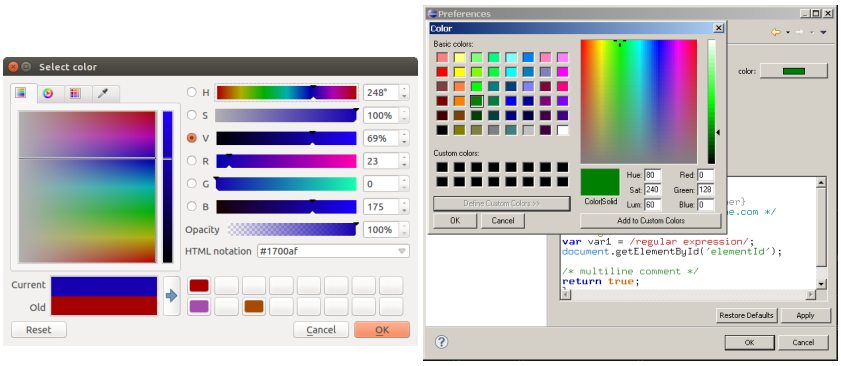

Figure 4.2: The QGIS and Windows color pickers.

On the left, figure 4.2 shows a color picking tool found in the open source geographical information system named QGIS. It is a typical

example of a widget recreated in many different apps. We imagine a user that has gotten used to the particular layout, idiom and affordances of this tool and wants it to be their color picker of choice in some other context, such as formatting a text document, and wants to know how to achieve this. People unfamiliar with the nature of today's software ecologies would be surprised to learn that the answer to this question is that it is economically impossible. On the right, figure 4.2 shows the standard Windows color picker, invoked from the Eclipse code editor, which is written in the Java language. It is visually clear that the two widgets perform the same basic function, and that they are even capable of sharing the same representations for the color value[3].

[3] The range of useful color tools is in fact much more diverse. Jalal et al. (2015) study how expert designers invent new mechanisms and contexts to manipulate color, and argue that digital color manipulation has been wrongly reduced to a simple problem of selecting and storing colors in context-less palettes.

Our hypothetical user would like to replace the QGIS color picker for the Windows one. In a malleable paradigm, this should hardly be distinguishable from using the color picker in the regular way, i.e. there should be nothing preventing the user from sampling colors from or assigning them to the Eclipse environment from the QGIS color picker. There should also be a direct means to assign the favored color picker as the standard one to be spawned in the code editor, e.g. putting it in a virtual drawer in the Eclipse interface.

In practice, a naive user expecting the color picker to be a reusable artifact might discover that it cannot pick or drop colors outside the window of the host application. We could imagine them closing the host application expecting the color picker to stick around, instead of disappearing. These kinds of problems are an everyday annoyance of modern interfaces, causing expert users to invent various idiosyncratic workarounds to carefully mediate between apps. For example, our user might use the host app of the color picker as a transit station where colors to be sampled are imported by taking a screenshot, and exported by copying and pasting e.g. the HTML notation for the color.

Barriers to reuse originate in development choices outside even the expert user's control. Our example color picker originates in an app, whereas a luckier user might have access to and prefer a color picker hosted at the operating system level, which may travel between different applications. On a deeper level, these problems are embedded in the stack of technologies used to implement and run the color picker.

Given that our user is or has recruited the assistance of a developer, the first road block to performing this adaptation is to penetrate the inside of a delivered application to locate the color picker to-be-substituted. Second, the developer must reconcile the form of the migrating color

picker with the expectations of its new environment. Added to the difficulties created by the mismatch in implementation languages (C++ on the left, and a mix of Java and C on the right), these come together with different idioms for interpreting the contents of memory, and even for addressing and painting areas on the screen. Even in the case where the color pickers are implemented in the same language, the imported tool will likely have to be carefully modified to conform to API requirements, such as method names, parameter order, and data structures. It may be difficult even to circumscribe the parts of the source code that deal with constructing the color picker to be replaced.

While some heroic efforts might succeed in transplanting this color picker, the resulting assembly would be of little value. The user would end up with two distinct copies of the color picker that would remain insulated from each other, i.e. the user would still be unable to use the same tool to pick a color in QGIS and paste it in Eclipse. More critically, the user would effectively have taken on the responsibility for maintaining their own personally customized version of the entire application In summary, the color picker has no economic reuse value across the app boundary, a poor compensation for the hundreds of hours of developer time that went into producing it. Figure 4.3 visually represents the hypothetical process of porting the color picker.

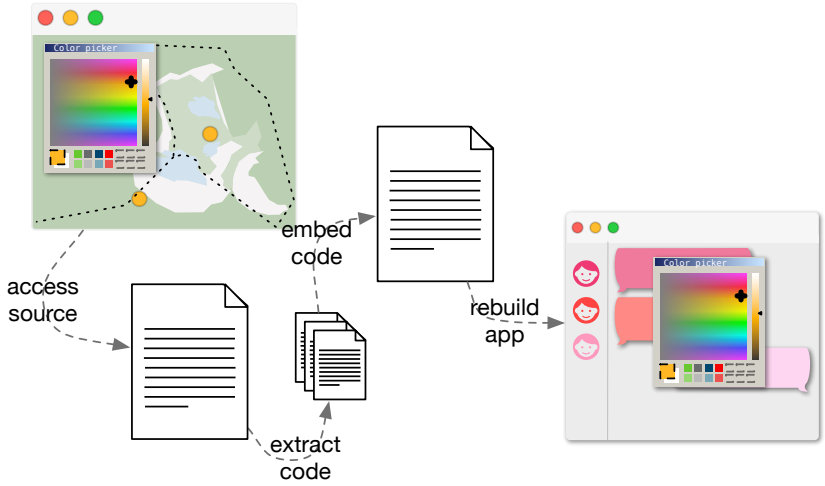

Figure 4.3: In contemporary apps, transporting an interface elements requires first reversing the development process at the source, and then re-iterating it at the destination. This process is not only exceedingly complicated, the resulting artifact is also likely unsupportable in the long term.

Large-Scale Scenario: Tailoring iNaturalist

In the second scenario, I consider how malleability interacts with the presence of a larger community of users and developers. This scenario is based on our case study, drawing in hypothetical adaptations to iNaturalist described by AS.

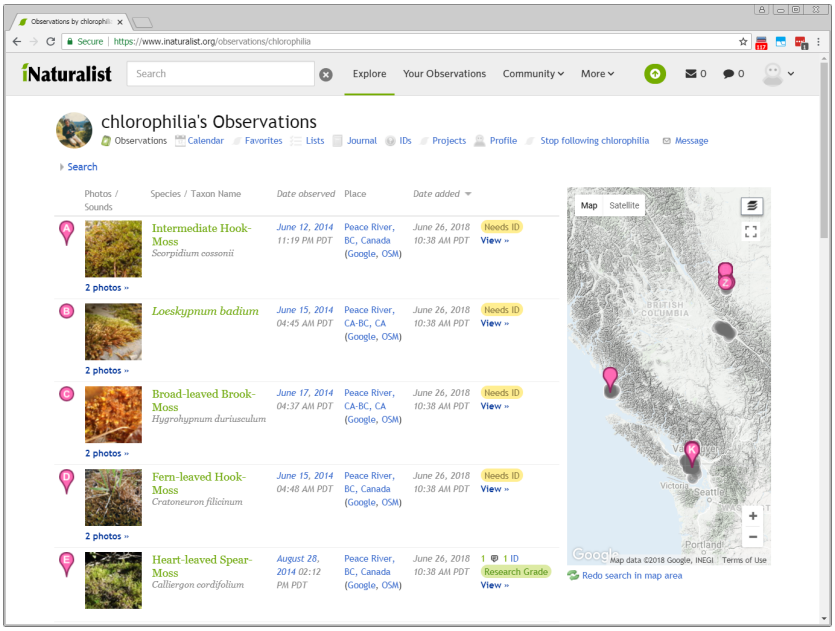

Figure 4.4: A view of one contributor's observations on iNaturalist

Figure 4.4 shows a typical view of the iNaturalist web app in use, specifically the first page of a user's contributed observations. It is an app written mostly in the Ruby and JavaScript languages. Arguably iNaturalist should offer excellent conditions to users who wish to adapt it for their idiosyncratic needs, considering that it is open source, developed to serve the public good, well-maintained, and has a welcoming community. In our example, AS, an expert user, wishes to adapt this existing platform to assist their own projects by making a number of additions to its functionality and interface. He has a general familiarity with the popular statistical analysis packages used in his field, and an everyday acquaintance with web technologies, but is certainly far from the skill-set of a professional developer.

AS wants to adapt the iNaturalist platform in (at least) the following ways:

- Users should be able to contribute lists of species that are considered to be available in particular locations, and to associate those lists with geographical boundaries drawn in a free-form way on maps;
- Project owners should be able to assign project members as curators of these lists;

- The existing representation of species should contain fields beyond those currently offered, such as their natural habitat and descriptive characteristics (e.g. "bush-like");
- New views should be added next to the existing ones for contributing and viewing observations, such as a page to select two or more lists and visualize their intersection or difference, e.g. as a sunburst diagram.

In a malleable paradigm, these adaptations should be feasible to carry out with the help of a developer with minimally-privileged access to iNaturalist's source code. Ideally, a programmer should be able to create an artifact similar to our malleable color picker for each distinct feature, from scratch or by tailoring an existing interface element extracted from iNaturalist or from elsewhere. These artifacts should then be possible for any end user to overlay onto their personal instance of the interface.

Let us imagine that AS has access to the resources of an expert development team to implement the improvements for his community. The hypothetical realistic process of adapting iNaturalist follows a similar procedure to the previous example: determining the sites where changes are to be made, then reconciling existing expressions with the new features we want to add. In figure 4.5, we show the root page of iNaturalist's source code, as shown in GitHub.

As in the previous scenario, there is total lack of correspondence between the source code and interface in use shown in figure 4.4. This isn't just a superficial phenomenon – there is no reliable way to determine, e.g., given the view of the "Intermediate Hook-Moss" observation in Figure 4.4, which part of the source code in one of the numerous deeply nested directories shown in Figure 4.5 is responsible for it. This does not owe to the particular technology choices made by iNaturalist's developers – it is the routine condition of the dominant software development paradigm. Experienced programmers take this discrepancy for granted, and learning to navigate it is a core skill of large-scale software development. This is typically done by in-depth study of the structuring conventions used by the frameworks and communities at hand, trial-and-error programming, reading long stack traces, and searching the source code for tell-tale strings, etc.

This scenario presents issues beyond the difficulty of simply expressing the desired tailoring. It brings up community-scale issues such as how new functionality is to be shared, how it is to be developed and

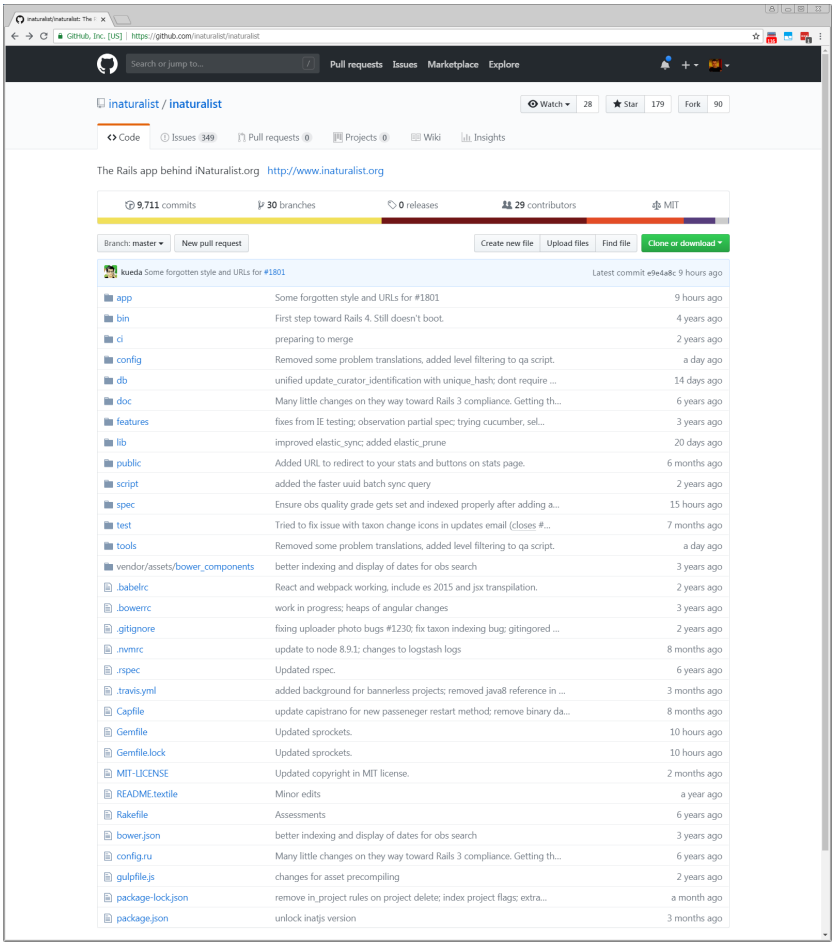

Figure 4.5: The source code for iNaturalist in GitHub

tested, and at what cost, and how it is to be deployed. AS's needs for tailoring in this case go beyond those of a single person desiring a personalized color picker. In this case, not only should the new functionality be available to members of AS's biodiversity project, but it should also be possible for other communities to opt into his custom additions, and for multiple authors' additional functionality to be combined. This requirement sums up the notion of software development for plurality, i.e. "multiple, located, partial perspectives" (Suchman, 2002). AS recognizes and wants to plan for the fact that unrelated communities may have their own perspectives on what species exist and which aspects of them are relevant to their practice. These all need to coexist with the "base function" offered by the core iNaturalist platform itself, and its base community. How this is done in practice has implications for the networks of cooperation that are likely to emerge, e.g. which communities are likely to share data.

A typical design solution to problems such as this are to add more and more functionality to an app over time, resulting in interfaces similar to the "giant ribbon" familiar to users of modern office applications, in which menus and toolbars are packed with tools represented by tiny icons. This phenomenon has been named *software bloat*, and HCI researchers have developed multiple tactics for curbing it, e.g. automatic and user-controlled adaptation of interfaces (McGrenere, 2002).

Realistically, the adaptations will not appear fully formed, but will have to be developed iteratively in the context of the existing code base and user community. While simple prototypes and mockups might be suitable for early stages of this process, it is inevitable that later stages of design will have to be evaluated around patterns of real use. This implies that it should be possible for an adapting community to effectively sequester a running version of iNaturalist, and deploy it to users who have opted into participating in this design process. There are broadly three strategies available to manage this. Unfortunately, it turns out that none of them offer favorable economics.

First, the adapting community could attempt to create a tailored iNaturalist that shares as much code and infrastructure with the existing platform as possible. Modern distributed source control systems such as git[4] let the adapting community fork the platform's source code to create a separate working copy. Beyond this point, however, the costs start to escalate rapidly. To actually run their version of iNaturalist, they will need to become familiar with its deployment and integration pipeline. They will also need to figure out how to share data with the core community, itself a source of many breakdowns found in the case study. Unless they want to commit indefinitely to maintaining their adapted version, they will need to build a strong relationship with the upstream community, and become familiar with their coding guidelines, administrative practices, and ensure that their adaptation is structured in terms that they are prepared to accept and maintain. In practice, it is more likely that the two versions drift out of compatbility, as in the example of Natusfera (section 3.2). A contributing factor here is that adaptations to the forked source code are represented by a "diff", a set of textual additions to and removals from the code. Diffs can be used to apply the adaptation to other copies of the source code, or to rewind to previous versions. However, it is cumbersome, and often impossible, to combine diffs from multiple sources, e.g. the respective changes of the iNaturalist and Natusfera teams.

[4] `git-scm.com`

Second, the adapting community could choose to develop a new system, replicating a large proportion of iNaturalist's function and data

model, ignoring elements of the interface not relevant to their needs, and implementing their desired features. The integration with iNaturalist would then occur via its published APIs – e.g. observations in the fresh system would be matched one-to-one with observations in iNaturalist's database, user accounts created on the new platform would be mirrored in the other, etc. The development costs of this new system would be extreme, and in practice integration via the API will likely be inadequate. This was P6's experience in the example seen in section 3.2.

Finally, the adapting community could build a long-term working relationship with the core development team, and persuade them that their features should be added to the platform. As expressed by AS, this path is ideal in that it allows the platform and his community to mutually benefit: *"If there was ever funding available, then it would be appropriate probably to talk to the makers of iNaturalist and say 'We have this project that kind of dovetails or parallels with what iNaturalist is doing, what aspect of it do you think does gel with what you're trying to do?'."* However, this scenario is unlikely, considering iNaturalist's historical approach of pursuing a narrowly scoped design so as to not spread themselves too thin.

4.2 Discussion of scenarios

The following problems came up in the scenarios:

- Interface/code correspondence: typical programming environments do not help adapters trace graphical elements and their behavior to the code producing them;
- Untangling interface code: code defining interface elements is difficult to re-contextualize because it is coupled to the larger system through references to its entities and their APIs;
- Externalizing adaptations: different forms of adaptations, such as diffs and plugins, have different levels of support for distribution, sharing, combination, and further adaptation;
- Federating development: existing tools such as version control systems do not help communities adapt a shared system in different directions without breaking interoperability.

The problems of pluralistic data and installations that I briefly touched

upon in section 3.4 also came up peripherally in the second scenario.

Malleability and software engineering

Our scenarios deal with the key problem of software evolution, incorporation of new user requirements and changes in the software environments. They re-iterate the conclusion that the cost of adapting programs is, in general, proportional to the complexity of the adapted systems rather the complexity of the desired features. However, the relations of production implied by our scenarios is different from those reviewed in section 2.6.

Malleable software continues the trajectory from centralized to pluralistic software development, from Parnas' central management (1971) through Bennett and Rajlich's staged model (2000) and Kell's "growing islands of functionality" (2009). The key addition is that malleable software conceptualizes user interfaces, traditionally the "end product" of software development, as further sites for adaptation and integration. It can be seen as a unification of today's highly distributed, broadcast model of development and the local, social development networks seen in the tailoring studies reviewed in section 2.5. Basman et al. (2018a) have named this model of software production *open authorship*. They visualize this as an open-ended and continuously expanding "graph of authors" creating, adapting, combining, and using programs. Open authorship and malleable software are sympathetic frameworks, the former starting at the level of programming languages, and the latter starting at the level of user interfaces.

Basman et al. find three key properties to be conducive to open authorship. First, adaptations to a system should be additive rather than destructive, i.e. they should be performed by creating a new module and attaching it to the existing system, rather than rewriting existing code. Additive modification is useful because it lets the adaptations of multiple developers be aggregated, whereas multiple modifications of a text may not be possible to consolidate. Second, systems should be freely and stably addressable, meaning that every part of a program can be referenced from outside the program, e.g. by using a global path expression, and that these references are likely to remain valid. Addressability is key to increasing the expressive power of additive modifications: it is the property that allows new modules to extend properties that are deeply embedded in the source code of an existing system. Third, all elements of a system should be externalizable, meaning that it should be possible to represent and share them as data, e.g. as text documents. This property is necessary to enable use cases

such as transporting a behavior from one system to another. In the next chapter, I discuss related work and the design of a programming model for malleable software in terms of these properties.

Researchers in the end-user development (EUD) (Paternò and Wulf, 2017) community have developed several tools that could help communities adapt and maintain malleable software. For example, Ko and Myers (2008) have created WhyLine, a tool for tracing system behavior to code by selecting questions from a menu such as "Why did this value become undefined?". Wulf et al. (2008) review experiments with several programming tools that help users reveal and modify the program components making up their interfaces. Such tools would likely be essential artifacts in a mature malleable software ecosystem.

Malleability and interaction design

I have described an aspirational division of labor between end users and (local and remote) developers. End users should be able to bring together interface elements and documents, and to some extent these should be able to "just work", as in the color picker scenario. This implies that a malleable software environment has some capacity to recognize the presence of interface elements that should interact, and to instantiate the necessary machinery. It is an open question how much typical API matching requirements can be relaxed by computational means. In cases where elements cannot be automatically coordinated, it should be possible for a local developer to assist a non-programming user by adding some code to align the elements the user is trying to bring together. Once such an integration has been performed, it should be possible for the user to keep and share it.

The exact line between adaptation tasks that require programming and those that do not is a critical usability factor in malleable systems. For example, Edwards et al. (2009) review a decade of work on an experimental architecture for *recombinant computing*, a framework for ubiquitous computing that is somewhat similar to the vision of malleable computing. The Obje architecture created by Edwards et al. (2009) lets users combine devices such as printers, projectors, and laptops through a PDA interface. They conclude that allowing ad hoc interoperation of devices inherently creates an added cognitive burden on users compared to off-the-shelf interfaces, because users must establish useful semantics for their devices. To reduce this interpretive burden, they create a set of reusable templates that encapsulate routine configurations of devices, called *setups* (Newman et al., 2008).

However, what is a routine configuration highly depends on the particular community of practice in which the configuration takes place. Therefore, such templates for interaction should be possible for communities of users and programmers to create and adapt. This should allow them to collaborative shoulder the interpretive burden of design and create a local repository of useful interactions. This hypothesis is supported by several of the examples of community tailoring reviewed in section (2.5): in both *Buttons* (MacLean et al., 1990) and X Windows (Mackay, 1990a), a key faculty enabling tailoring is that users can externalize their personal customizations, as buttons and configuration files, respectively.

Some researchers have developed interaction techniques that let users modify their systems by acting directly on the interface. These demonstrate interesting end-user adaptation techniques in addition to the drag-and-drop adaptations I have focused on so far. Grab n' Drop (Eagan, 2017) is a malleable toolglass (Bier et al., 1994), which can be used to "grab" buttons, e.g. from conventional toolbars, transforming them into toolglasses. Façades (Stuerzlinger et al., 2006) provides direct manipulation techniques for splitting and combining interface widgets. For example, Façades lets a user gather their preferred tools in a graphics editing app into a custom floating palette. Improv (Chen and Li, 2017) lets users extend existing interfaces with cross-device interactions by demonstrating them. For example, Improv lets users adapt a video player to be controlled via smart watch gestures by first clicking the play/pause button, then demonstrating a gesture to associate with that button. EmbeddedButtons (Bier, 1992) merges UI elements and document content, letting users create and modify buttons by modifying picture and text markup tags, which represents behavior in the same format as graphical properties such as font weight and color.

In this chapter I have described how code and interfaces currently have unsuitable properties for malleable software. In the next chapter, I survey related work to find alternative, positive properties for malleability. I then develop a programming model that attempts to combine these properties.

4.3 Key Takeaways from Chapter 4

The design vision of malleable software encompasses challenges for both software engineering and interaction design.

From a software engineering perspective, malleable software requires languages and tools for *open authorship* (Basman et al., 2018a). Open authorship expands the goals of flexible and reusable programming to encompass program transformations that are highly unusual in current software development practice and research. If malleable software development is to be economically feasible, an adaptation created relative to one program should be applicable to other programs and result in a similar change in functionality. The inverse of this should also be possible, i.e. developers should be able to circumscribe part of a running system and create an adaptation that can move the circumscribed functionality to other systems.

From an interaction design perspective, malleable software requires design principles and interaction and tailoring techniques for interfaces that can be pulled apart and (re-)combined at runtime. Users of malleable software should not have to do ongoing work to assemble the interfaces they use, but should provide hooks for adaptation work to be done where necessary. Based on previous work on tailorable software and design-in-use, I emphasize that such techniques should enable the work of assembling appropriate interfaces to be distributed among users and developers.

5

Entanglers: A Programming Model for Malleable Software

Both conventional UIs and state-of-the-art UIs designed to be adaptable fail to live up to the requirements of malleable software. A key weakness is in their handling of interactions between system components. Interactions are generally too brittle and too expensive to adapt to feasibly support malleable use cases. I present a programming model that improves on these qualities by representing interactions as entanglers*, declarative descriptions of the pre-conditions and mechanisms necessary to instantiate interactions. Entanglers enable a* co-occurrence engine *to recognize and construct interactions dynamically, and enable programmers to adapt systems with new interactions. I describe Tangler, a proof-of-concept implementation of this model, and demonstrate its power and generality with two adaptation scenarios.*

In the last chapter, I argued that conventional programming tools are ill-suited for malleability for multiple reasons. Programming languages, software architectures, and development tools limit how user interfaces can be combined, and how programs can be adapted, both at the scale of individual users and distributed communities of users. In this chapter, I focus on the specific issue of how programs can express and execute *interactions* under the conditions of malleable software. By interactions, I mean the relationships that connect elements such as input devices, cursors, scrollbars, buttons, and document content to each other. These relationships are made up of control structures such as event listeners, state machines, and reactive functions.

In section 4.2, I argued that malleable systems should have some facil-

ity for automatically coordinating interface elements with their changing surroundings to guarantee a smooth user experience. This can be reframed as a requirement for interactions that are "self-assembling", in the sense that once a programmer has created an interaction, the system architecture is responsible for recognizing where and when they should be instantiated. This process of automatic assembly should enable the basic use case where users move their tool between different kinds of digital documents (figure 5.1).

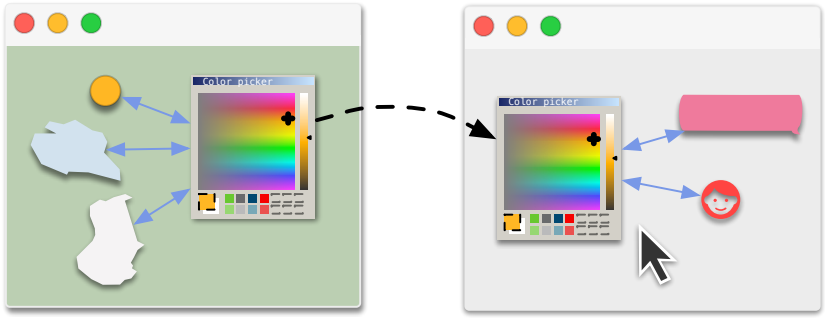

Figure 5.1: Interface elements should be continuously re-connected as the preconditions for particular interactions occur.

Furthermore, in cases where the system does not have enough information to instantiate an interaction, or users wish to reconfigure how elements interact, it should be possible for a programmer to adapt the system to enhance the existing interaction or produce a new one. For example, consider a case where a user has a conventional color picker, operated by clicking a graphical object to select it and then clicking a color swatch to apply the color it holds. This user would like to reuse the existing elements with a different interaction technique. They would like to be able to drag the color swatch over shapes and then "click through" it to apply the color directly onto them, in the manner of a toolglass (Bier et al., 1995). The code enacting this adaptation should take an additive form, so that a system can feasibly aggregate many such adaptations as it evolves.

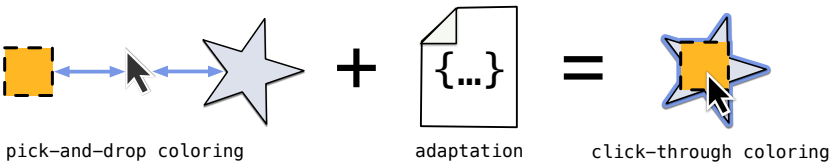

Figure 5.2: It should be possible to create or refine interactions among elements via additive adaptations.

To solve these issues, I deconstruct the notion of interaction into multiple concepts governing different steps of the life cycle of interactions, i.e. their design, instantiation, enaction, and destruction. In our model, programmers create *entanglers*, which are analogous to recipes for interactions, i.e. they describe potential interactions. Entanglers define

the pre-conditions for an interaction in a *co-occurrence description*, and the control structures that should be constructed when those conditions are met in an *entanglement template*. When users create the conditions for an interaction to be instantiated, the system detects a *co-occurrence* of concrete elements, and instantiates an *entanglement*, and when the conditions cease to be met, it is torn down.

In this chapter, I present a programming model based on these concepts. Antranig Basman, Simon Bates, Michel Beaudouin-Lafon, and Clemens Klokmose contributed to the development of this model. First, I review previous user interface systems research that has implemented aspects of malleable software. I then present the key concepts of our programming model in section 5.2. In section 5.3, I present TANGLER, a prototype implementation of the model as an *integration language* (Kell, 2009). In section 5.4, I demonstrate TANGLER's capacity for adapting interactive systems through two in-depth examples. I discuss our model in relation to other programming mechanisms in section 5.5, and conclude with a summary of our contributions and a brief summary of future work.

5.1 Related work

I review various techniques and systems for making software more malleable, i.e. creating interfaces made up of parts that can be taken apart and combined to achieve some desired composite functionality.

Plugin architectures

Sometimes designers explicitly plan for a limited version of the kind of reuse we saw in the malleable software scenarios (section 4.1). For example, they create plugin architectures that enable programmers to extend apps with additional functionality.

Plugin architectures are widely adopted in today's software ecosystems, e.g. every major browser supports plugins, also called extensions. Typically, the architecture provides access to an app's document manipulation and interface-drawing API. It also requires facilities to execute plugin code safely. Chen (2019) provides a good example of the design and implementation choices to be made in creating a modern plugin architecture.

Plugins have somewhat successfully allowed users to add and replace

functionality without programming, e.g. by installing a package from a curated repository. For instance, at the time of writing, the Firefox browser's plugin repository features at least nine different color pickers. Furthermore, multiple communities have invented community-curated packs combining multiple plugins that work well together (Haraty et al., 2017).

However, browsers clearly do not provide for the kind of malleability presented in the scenarios. In practice, plugins form an outer layer of customizable functionality around a core of base functionality. In the case of browsers, both the browser interface and the interfaces provided by web pages are in the core. Even though a browser extension can in practice modify the content of any web page, this gives it only limited power to modify the user experience, because web apps are not malleable. For example, an extension could be used to change the appearance of observation pages on iNaturalist's website, but it could do nothing to change the system's internal notion of what kind of data is in an observation. This can be described as a lack of addressability, in the sense that the majority of code defining the functionality of browsers and web pages is unadressable by plugins.

Furthermore, a plugin written for one platform is likely incompatible with any other platforms. Therefore, plugin architectures result in strongly bounded reuse, where creators and users of plugins commit economically to one particular environment over others. Notably, there are real-life cases of popular features that different developers often choose to recreate in their chosen environments. For example, many code editors have plugins to replicate the characteristic key bindings of the ubiquitous vim editor[1], and all the mainstream web browsers have several different ad-blocking plugins.

[1] vim.org

This situation, where each platform comes with its own conventions for extending the interface, is reminiscent of the early days of electrification, and similar infrastructural technologies. In hopes of dominating the market, proponents of alternating and direct current raced to spread adoption of their chosen standard, increasing the value of their infrastructures and locking out competitors (David and Bunn, 1988).

Black-box interface extensions

HCI researchers have created several toolkits for plugins that can be applied to legacy software, i.e. systems that do not have plugin architectures. These black-box interface extensions (Eagan et al., 2011) can modify the interfaces of apps without access to their internal structure

or source code.

Toolkits in this tradition augment legacy apps by interfacing with them at various levels. Scotty (Eagan et al., 2011) lets programmers create additional interface elements for OSX apps by injecting code through the Cocoa UI toolkit. Interface Attachments (Olsen et al., 1999) and Façades (Stuerzlinger et al., 2006) modify features of the interface of running applications through the windowing system. Improv (Chen and Li, 2017) creates cross-device interactions in the browser by transforming input events on remote devices to simulated input events on existing interface elements. Prefab (Dixon and Fogarty, 2010) uses computer vision to recognize and modify interfaces at the pixel level.

Black-box interface extensions are powerful because they create the same sort of malleability as traditional plugins, but reduce their compatibility issues. However, their purpose is to enhance, reorganize, or integrate existing systems. Therefore they are limited by the features of the apps they target and cannot be used to build systems from the ground up, nor disassemble and repurpose parts of a system.

Interface embedding

Several mature software platforms have provided mechanisms for sharing functionality across apps by embedding interfaces and documents inside each other. Microsoft and Apple have both developed so-called compound document standards for their operating systems, respectively named OLE[2] and OpenDoc[3]. These let users embed, e.g., content from a spreadsheet in a word processing document. With OLE, embedded content can be used as a hyperlink to its "home app", which will be opened when the user interacts with the embedded content. With OpenDoc, the app's menu bar changes to present the appropriate set of commands when embedded content is selected.

[2] docs.microsoft.com/en-us/cpp/mfc/ole-background

[3] web.archive.org/web/20071009214704/http://www.sundialsystems.com/articles/opendoc.html

Several conceptually similar web technologies exist, e.g. the HTML iframe element[4] and the Java Portlet standard[5]. Both let several web services be aggregated on one page. iframes do this by simply displaying linked pages in a window, whereas Portlets package services in a finer-grained way, creating a composite document that can, e.g., be styled in a uniform way. These technologies have been used to create web mashups (Hartmann et al., 2008) and portals (Zhou, 2003), web pages that combine tools created by different developers.

[4] w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html

[5] jcp.org/en/jsr/detail?id=162

Another related mechanism is embeddable widgets. Some websites provide snippets of markup that users can copy and paste into their

personal websites to embed some functionality. For example, in our case study, AS used a tool on the iNaturalist website[6] to generate a widget that displays the most recent observations in his area.

[6] inaturalist.org/observations/widget

The ability to copy and paste interface elements is a key affordance in the kinds of reuse stories I imagine. It is an example of the value of externalization. However, mixed interfaces created with the surveyed techniques are severely limited because embedded elements are completely isolated from each other. As a result, they cannot support either of the example problems described in the introduction to this chapter. Effectively, they re-create the siloing of apps at a finer granularity.

Live object systems

Live object-oriented programming (OOP) systems in the tradition of Smalltalk (Ingalls et al., 2016) bear strong similarities to the vision of malleable software. In these systems, each distinct object on the screen is associated with an object in the virtual machine, which can be accessed and edited by users to change system functionality and appearance. The *Buttons* (MacLean et al., 1990) system that I have reviewed previously is an exemplar of this paradigm. In the following, I treat Lively (Ingalls et al., 2016) as a representative modern example.

Live object systems provide a convincing solution to the problem of correspondence between interface elements and code. Lively uses a version of the Morphic user interface toolkit (Maloney and Smith, 1995), which was designed according to the principle that *"the thing on the screen is the actual thing"* (Ungar and Smith, 2013). They achieve this by making all objects on the screen, called morphs, define their own representation, state, and behavior. This means that the general workflow for adapting morphs is very direct: users can point at UI elements with an inspection tool, locate the appropriate slots, and type in new values and procedures. They can also freely reorganize morphs, e.g. to collect frequently used buttons in a toolbar.

However, the live object paradigm presents some fundamental problems in how it deals with interactions. Lively objects interact through typical object-oriented techniques, such as method calls, aggregation, and inheritance, i.e. interactions are defined inside and across morphs. Systems behaviors defined by code across multiple modules are called cross-cutting concerns (Kiczales et al., 1997). Cross-cutting concerns can result in excessive coupling and scattered code.

Excessive coupling results in brittle interactions. In a Lively version

of the color picker scenario, it would be easy to move the color picker from one window to another, but it would very likely be non-functional once removed from its original context (figure 5.3).

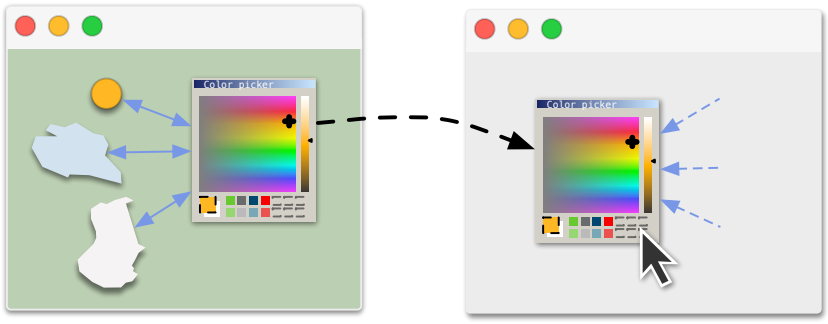

Figure 5.3: In environments based on live objects, such as Lively, it is possible to move interface elements between contexts, but the relationships connecting them to their environments are likely to break in the process.

Scattered code makes adaptation more difficult and error-prone. For example, consider the adaptation example from the introduction to this chapter, where a programmer wants to create a click-through interaction combining a mouse cursor, a color swatch, and a shape object. This leads to implementation trade-offs such as choosing which objects should be extended and whether to modify their existing API or add new methods. In this case, the additional behavior could in theory be implemented by extending the swatch object, but would require extending its responsibilities to include acquiring target objects and applying colors to existing objects, as well as adding guard code to distinguish whether it is clicked to select a new color or dragged to apply a color to objects in the canvas.

These issues are relatively mild when systems are developed in a closed world with a very stable object graph. They are particularly troublesome, however, in malleable systems, where objects are expected to come and go continuously.

Lincke et al. (2011) have created the context-oriented programming (COP) library ContextJS[7] to mitigate this kind of issue in Lively. COP builds on OOP by providing dedicated language abstractions for defining and composing adaptations to basic program behavior. The most well-developed COP abstraction is the notion of a *layer* that encapsulates adaptations to multiple modules (Costanza and Hirschfeld, 2005). Layers can be dynamically activated and de-activated, and combined with other layers. Our model bears some similarities to COP concepts, which we discuss in section 5.5.

[7] `hpi.uni-potsdam.de/hirschfeld/trac/Cop/wiki/ContextJS`

Webstrates

Webstrates (Klokmose et al., 2015) is a web server that persists and synchronizes the DOM content of web pages that it hosts, so that the same content can be accessed and modified by multiple users and devices concurrently. This means that web pages running on Webstrates are turned into what-you-see-is-what-I-see (WYSIWIS) collaborative interfaces (Stefik et al., 1987). Since web pages can contain scripts and embed and modify other web pages, they blur the distinction between documents and apps. Klokmose et al. call such pages web substrates, or webstrates.

Klokmose et al. demonstrate Webstrates' support for malleability at two levels of granularity. First, they show multiple users connecting their personal word processing webstrates to collaborate on a shared document. This is achieved by making each user embed the document webstrate through a mechanism called transclusion (Nelson, 2012). Figure 5.4 illustrates a network of webstrates transcluding each other. Second, they show users collaboratively modifying their personal webstrates' interfaces by extracting a button from one webstrate and inserting it in another. This is possible because parts of a webpage can easily be externalized as HTML markup, which is immediately rendered when inserted into another page.

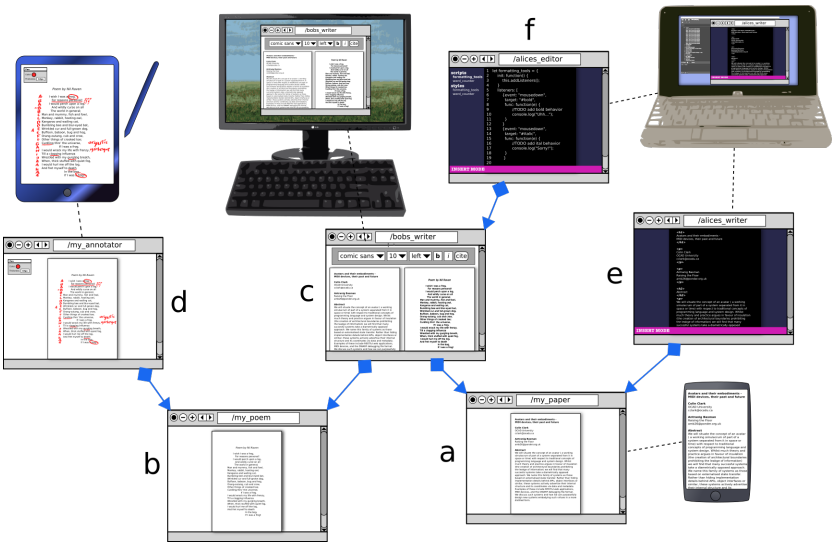

Figure 5.4: An example of a network of interconnected webstrates. The blue arrows are transclusions establishing application-document relationships. a) A paper, displayed on a mobile device. b) A poem. c) Bob's word processor editing a and b on his desktop computer. d) Bob's annotation webstrate which lets Bob hand-annotate webstrates with his tablet and stylus. e) Alice's HTML markup editor, editing the paper a on her laptop. f) Alice's script editor, extending Bob's word processor on a different window of her laptop.

However, Webstrates relies on conventional code with the same coupling and code scattering issues that apply to live object systems. Malleable tools like personalized editors and transportable buttons embed strict assumptions about the structure and naming conventions of the

webstrates they can operate on. Therefore, webstrates created by different developers cannot be freely recombined, because they are likely incompatible with each other. Reprogramming these elements to suit new contexts is difficult for the same reasons.

Control Structures for Programming Interactions

Toolkits and languages for developing interactive systems have introduced a diverse array of control structures for interactions, including event listeners, state machines (Appert and Beaudouin-Lafon, 2006; Blanch and Beaudouin-Lafon, 2006), constraint rules (Sutherland, 1963; Myers, 1990), trigger-action rules (Ur et al., 2016), reactive data-flow functions (Czaplicki and Chong, 2013; Elliott and Hudak, 1997), and processes (Magnaudet et al., 2018). These are usually motivated by creating better expressive match (Olsen Jr, 2007) between code and desired interactive behavior, as well as making the code more maintainable and less error-prone (Myers, 1991).

These mechanisms are developed under the assumptions of rigid systems where interactive behaviors are expected to remain stable at runtime. Their run-time form is generally highly divergent (Basman et al., 2016), i.e. they cannot be targeted for modification at run time unless the original programmer has seen fit to expose references to them. Therefore, they tend to introduce behaviors and couplings among elements of the system that cannot be undone by subsequent adaptations. In TANGLER, we implement event listeners, state machines, and model relays as a set of reasonably expressive control structures. Our implementations are improved for open-ended development by giving each control structure a declarative form that is addressable and externalizable, i.e. they can be located at paths from anywhere in the program, and they expose all references to the components they orchestrate.

Integration authoring tools

Several HCI and programming language researchers have independently proposed decoupling interactions from interacting elements. Following Kell (2009), I call these first-class interactions *integrations*.

ICon is a toolkit for creating integrations of input devices (Dragicevic and Fekete, 2004). These integrations, called *configurations*, are expressed in a visual wire diagram language, where input devices are connected to virtual controls exposed by apps. ICon can be used to prototype novel configurations of input devices and adapt existing systems for accessibility, e.g. helping a user with hand tremors use

a mouse with a configuration that filters out small mouse movements in the data sent to apps (Dragicevic and Fekete, 2002).

OSCAR (Newman et al., 2002) is a system that lets users construct reusable configurations of home media devices such as speakers, screens, and microphones to accomplish tasks such as seeing who is at the door or playing music in their kitchen. It is built on the Obje framework for recombinant computing (Edwards et al., 2009). Integrations created with OSCAR are called *setups*. A setup that relies on having access to a set of devices, e.g. a webcam and a screen, will work with any webcam and screen. However, setups are based on streaming information from sources to destinations, i.e. transmitting media, rather than creating interactive relationships.

The PalCom architecture for multi-device interactive systems includes a similar concept, called *assemblies* (Svensson Fors et al., 2009). Assemblies define a set of dependent devices and services, and contain a script coordinating those devices by receiving and sending messages between their service APIs.

Entanglers are conceptually similar to these types of integrations, in that they are reusable integrations created by programmers. The key difference is that users or programmers must explicitly invoke configurations/setups/assemblies before they are used. This is appropriate when integrations connect somewhat stable ensembles of things, e.g. configuring a multi-device installation in a lab or setting up a projector for a presentation. It is less appropriate, however, at the granularity of our use scenarios, because it is unworkable to have end users select and activate an integration every time they move a widget from one environment to another. Instead, our integrations are invoked automatically as a result of particular configurations of elements, e.g., cursors are automatically created in response to mice being plugged in.

5.2 Conceptual Model: Co-occurrences and Entanglements

Our conceptual model distinguishes between the pre-conditions that must be met for an interaction to be enacted and the actual behavior of the integration. I call the former *co-occurrences* and the latter *entanglements*. This vocabulary is inspired by the metaphor of quantum physics.

In quantum physics, two or more particles can become entangled, meaning that their states depend on each other. Entangled particles are invisibly linked to each other even after they have been separated, until an external event breaks the entanglement. Similarly, in interactive systems, different elements are often linked by invisible forces, even when at a distance, such as the mouse and the on-screen cursor, the scrollbar and the document, or the "Cut" menu item and the currently selected text. I call the relationships among these elements *entanglements*. Entanglements are the logic connecting the elements that users perceive and act on. I use the word as a general term for the code that enacts interactions. In current systems, entanglements are typically implemented with control structures such as event listeners, state machines, and reactive functions.

In quantum physics, particles become entangled in a process of collision. In interactive systems, different types of such "collisions" can occur, such as plugging in a mouse, opening a document in a scrollable window, or selecting a span of text. I call these temporary groups of coordinated elements *co-occurrences*.

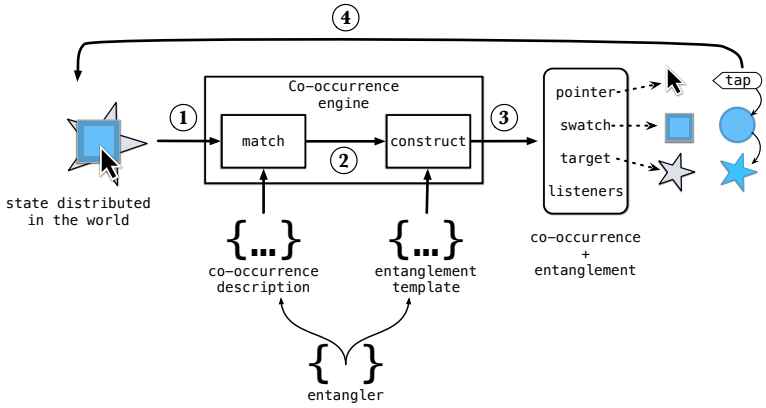

Figure 5.5: The procedure for creating entanglements from entanglers in our programming model, illustrated with the click-through color swatch interaction. 1: the state of elements in the system is continuously monitored by the co-occurrence engine. 2: entanglers describe a co-occurrence, the preconditions for an entanglement. The co-occurrence engine matches this description against the system state, and reports any successful matches. 3: entanglers contain a template of an entanglement, which combines the co-occurring elements with a behavior, enacting the desired integration. When the co-occurrence engine reports a match, this template is used to construct an entanglement with references to the co-occurring elements. 4: the entanglement complex drives a new interaction, causing further changes to system state.

This model allows us to deconstruct the process of constructing interactions. Rather than create entanglements directly, programmers write *entanglers* that describe the kinds of entanglements they wish to create along with the co-occurrences they require. The system run-time then automatically detects co-occurrences and create entanglements in response. I define a new declarative programming construct combining these two components. This construct is called an *entangler*, because it causes system elements to dynamically become entangled.

The first part of an entangler is a *co-occurrence description*, which describes a particular kind of co-occurrence and names its participating elements. The second part is an *entanglement template*, which describes

an entanglement combining the elements named in the co-occurrence description with one or more control structures to implement an interaction. The entanglers are managed by a system component called the *co-occurrence engine*, which matches co-occurrence descriptions and instantiates entanglement templates with the matched elements. Figure 5.5 illustrates the conceptual model with the click-through color swatch interaction from the introduction to this chapter.

Because the process depicted in figure 5.5 is automatic, our model can let users interact smoothly even as they add and remove documents, tools, and devices. Because entanglers are additive modifications and can react to any combination of state across a system, our model can support highly adaptable systems.

5.3 Prototype Implementation: Tangler

TANGLER is a proof-of-concept implementation of a programming environment based on co-occurrences and entanglements. It is implemented with web technologies (JavaScript, TypeScript[8], Node.js[9]) and runs in any modern browser. With TANGLER programmers can construct basic but highly adaptable graphical user interfaces, which I demonstrate in section 5.4.

[8] typescriptlang.org

[9] nodejs.org

An Integration Language

TANGLER adapts the declarative programming paradigm of Infusion[10], a framework for programming user interfaces that can be continuously adapted for people with different abilities and preferences (Basman et al., 2015). Infusion is an *integration language*, meaning that it lets programmers express relations between values rather than directly express general computation. This design goal follows Kell's proposal that the key to flexible software construction is to separate integration concerns, i.e. all the definitions establishing coupling among modules, into a specialized domain (Kell, 2009).

[10] https://fluidproject.org/infusion.html

We chose to build on the conceptual foundation of Infusion because co-occurrences and entanglements recognize and orchestrate configurations of state. Thus they are well-matched to a declarative programming paradigm, where "code" is itself a data structure that can be recognized and modified. TANGLER implements a subset of Infusion's syntax and semantics, and adds entanglers.

TANGLER programs are made up of components arranged in a tree structure that can be addressed through paths and queries similar to CSS selectors[11]. These selectors are a powerful addressing mechanism because they are less brittle than typical path expressions, e.g. whereas path address needs to specify the exact sequence of nodes leading to a referenced component, a selector can address "any component of type A that has an ancestor of type B". Because of this flexibility, references to component are less likely to become invalid as the component tree is modified and extended.

[11] https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Selectors

The two primary idioms by which components express relations among state are *model relays*, which express bi- or uni-directional relationships between the mutable state attached to components, and *option distributions* which use selectors to broadcast adaptations to other components in the tree. Option distributions let any component act as a mixin or mediator for other components, in addition to expressing local state and behavior. This makes TANGLER symmetric, in the sense that the same abstraction is used to define default behavior and adaptations to that behavior. Since components are declarative, integrations are inherently reversible, by destroying the integrating component, and adaptable, by injecting further options into it. Both operations can be performed at run-time, making TANGLER a live programming environment.

TANGLER also includes more traditional features to express outright computation and direct component interaction, such as event listeners, state machines, and invokers. Event listeners are the basic mechanism for responding to user input. State machines elaborate on event listeners by delegating to a different set of listeners depending on their current internal state. Invokers correspond to object methods, familiar from the object-oriented paradigm, but their side-effects are constrained to take the form of a mutation at a publicly declared path.

A component's authorial lifecycle starts with the definition of a type, analogous to an object-oriented programming class. A type definition is a JSON record containing the default values for the kind of thing represented by the component, and may have a number of parent types, whose defaults it inherits and extends. A component is generated at a particular path in the tree of components by a call to the framework function tangler.make, and they declare sub-components that are instantiated at the same time. The construction call or site may provide additional *options*, which overwrite or extend the default values given by the type. During the construction process, TANGLER evaluates selectors in components to determine where to distribute events, side

effects, and declared adaptations.

Entanglers

Entanglers are implemented as a component type that contains a co-occurrence description and an entanglement template. The description is a list of conditions naming and describing the elements of the co-occurrence and their state. The template describes the entangler's product as a component that may reference the names of co-occurring elements given in the description. When a co-occurrence begins, a copy of the template is instantiated, with the references substituted with mirrors of the concrete elements.

We have adapted Forgy's Rete algorithm for production rule systems (Forgy, 1979) to implement the co-occurrence engine. This algorithm turns currently active co-occurrence descriptions into a network where each node matches a single condition. The matching process is executed by sending the newest state of each component into this network. Rete minimizes recomputation by collapsing redundant conditions across different co-occurrence descriptions and storing partially matched data in the network. Our implementation is based on that presented by Doorenbos (1995). Whenever a new entangler is constructed, its co-occurrence description is added to the Rete network. The core TANGLER module reports any new or changed components to the Rete network. Both these operations cause the network to recompute any relevant conditions, possibly triggering a co-occurrence, which causes the Rete module to signal the main TANGLER module to construct a co-occurrence.

Rete expects conditions to either compare a component to a constant value or two components to each other. This respectively allows conditions to express intra-component state and inter-component state. The set of available conditions can be extended by registering new predicate functions with the Rete module.

Additional Modules

TANGLER features auxiliary modules to render graphics and handle user input (Figure 5.6). In a more fully developed environment, these could be the sites of further adaptation, e.g. to change the component representations. In our proof-of-concept system, however, we have decided to implement these modules as black boxes.

Components whose type is derived from viewComponent provide a draw

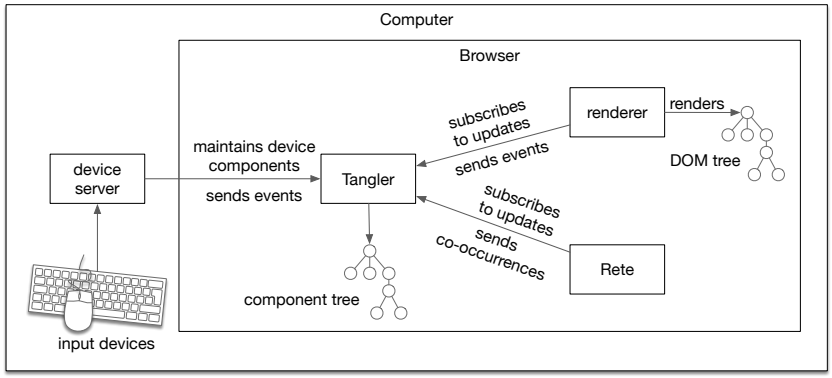

Figure 5.6: TANGLER and its auxiliary modules.

function that outputs a DOM element, and is used to continuously render the component by updating the web page running the TANGLER instance.

Input device events reach the component tree through two routes. Components representing input devices are managed by a Node.js server running on the user's computer, which users the node-hid[12] library to register Human Interface Devices (HIDs) and monitors their activity. The device server detects events by monitoring differences between successive device states, and forwards them to the device component. This server is necessary to support multi-mouse and bimanual interactions, because the browser client cannot query the operating system for HID data. Furthermore, an event funnel (Eagan et al., 2011, p. 228) on the web page consumes all events emitted on DOM elements and forwards them to the source viewComponent. To minimize confusion, TANGLER events are named with a prefix denoting their source, e.g. device-mousedown and dom-mousedown. This is a pragmatic choice to help us demonstrate how our programming model can be used to adapt input devices, not just GUI elements.

[12] github.com/node-hid/node-hid

Programming Example

We walk through the implementation of our running example, where a color swatch is adapted to become a click-through tool (Bier et al., 1995).

The initial color swatch lives in a toolbar area, and reacts to being clicked by revealing a color selection dialog. Figure 5.7.1 shows a representation of a component tree containing this toolbar, as well as a canvas area. Listing 5.8 defines the component type by calling the library method tangler.define, either from a JavaScript file or the browser

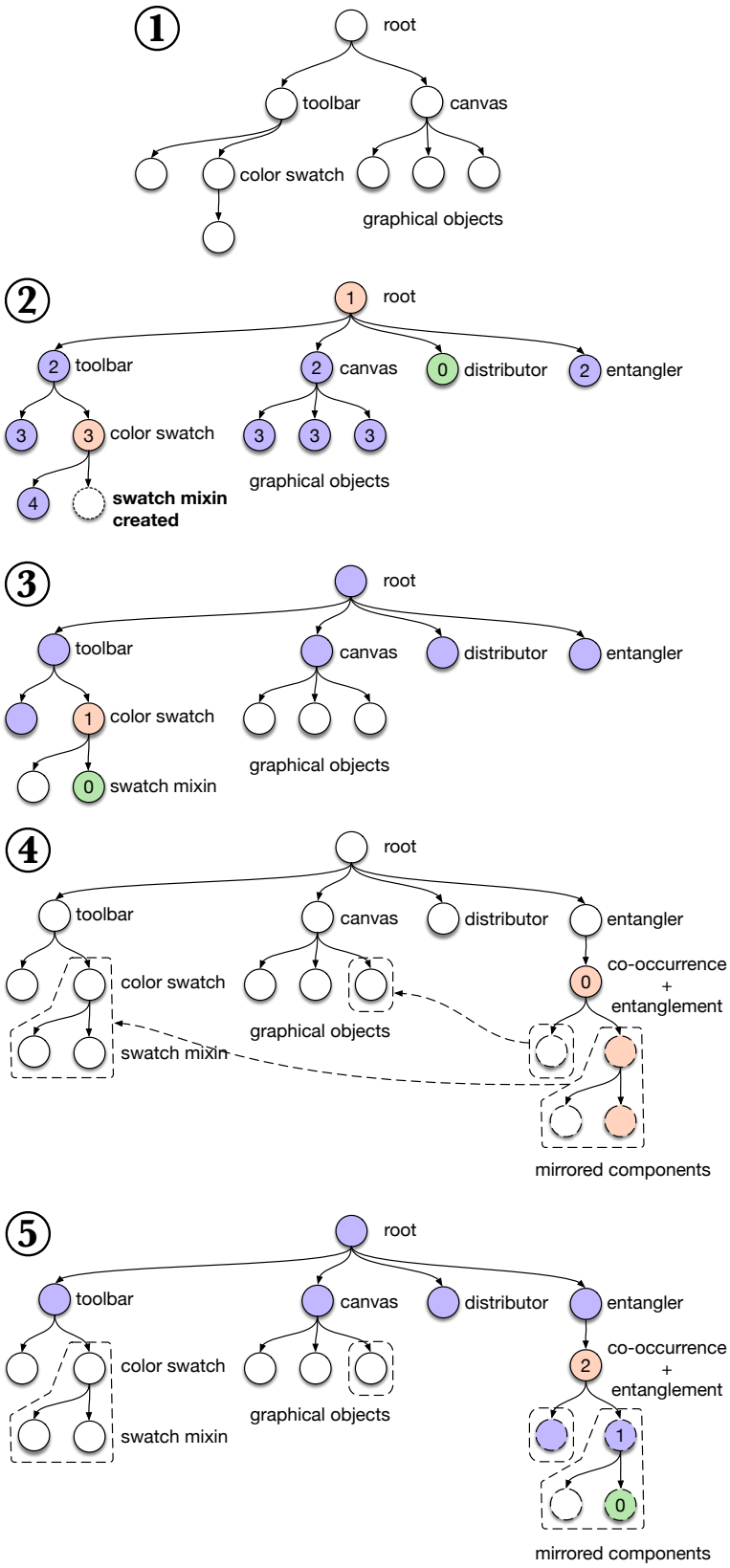

Figure 5.7: The development of the TANGLER component tree in response to the color swatch adaptation. Explanations in the text.

console. tangler.define stores a component definition under the given name. It can also be used to store pure functions that will be used by components, e.g. lines 24 and 49. We omit the definitions of such functions from the examples for space reasons. We use the convention that definitions with the prefix "tangler" are part of the "standard library", while user-supplied definitions are given the prefix "defs".

```
1  tangler.define('defs.colorSwatch', {
2    // descendants of viewComponent are
       rendered
3    parentType: 'tangler.viewComponent',
4    // define the component's mutable state
5    model: {
6      position: {
7        x: 0,
8        y: 0
9      },
10     fill: '#ffffff',
11     stroke: '#000000'
12   },
13   // declare any custom events this
       component can emit
14   events: {
15     activate: null
16   },
17   listeners: {
18     // when the rendered component is
         clicked,
19     // fire the activate event.
20     // '{that}' is the self-reference
         keyword
21     'dom-click.activate': '{that}.events.
         activate.fire',
22     'activate.impl': {
23       // apply styling to make the color
         selector visible
24       func: 'defs.getVisibleStyle',
25       changePath: '{that}.colorSelector.
         model.style'
26     }
27     // '.activate' and '.impl' are
         namespaces. Other
28     // listeners on the same events with
         matching namespaces
29     // will displace the ones declared
         here.
30   },
31   // create the color selector dialog as a
       child component
32   components: {
33     colorSelector: {
34       // We omit the defintion of defs.
       colorSelector for brevity
35       type: 'defs.colorSelector',
36       // the options record is merged into
        the color selector at
37       // construction. This is used to
       override default values,
38       // in this case making the component
        initially invisible
39       options: {
40         model: {
41           style: {
42             display: 'none'
43           }
44         }
45       }
46     }
47   },
48   // the draw function is used to render
       this component
49   draw: 'defs.drawColorSwatch'
50 });
```

Figure 5.8: Original color swatch

The first step of the desired adaptation is to make it possible to drag the color swatch. This corresponds to regular in-place object extension. In TANGLER, this step can be achieved by first defining a component to be used as a mixin that will be attached to the existing object. Listing 5.9 shows a mixin component that replaces the event listener on the color swatch with a state machine that delegates between clicking and dragging.

If the system is expected to only include a single color swatch, we might add the mixin directly by using the library method tangler.make. In case there may be several color swatches, we instead apply the mixin with an option distribution (listing 5.10).

Figure 5.7.2 illustrates how the distributor inserts the mixin by resolving the selector 'root colorSwatch'. The numbers indicate the order in which TANGLER searches for matches to the selector. First, it traverses

```
tangler.define('defs.colorSwatchMixin', {
  // descendants of modelComponent may contain mutable state
  parentType: 'tangler.modelComponent',
  stateMachines: {
    clickOrDrag: {
      // idle, armed, dragging are the machine's states.
      // keys inside states define transitions.
      // the name of the key is a reference to the event
      // that triggers the transition.
      // in this case, the events are emitted by the
      // '{colorSwatch}' component, i.e. the nearest component
      // of that type in the tree
      idle: {
        '{colorSwatch}.events.dom-mousedown': {
          action: '{that}.events.dragStart.fire',
          nextState: 'armed'
        }
      },
      armed: {
        '{colorSwatch}.events.dom-mouseup': {
          action: '{colorSwatch}.activate.fire',
          nextState: 'idle'
        },
        '{colorSwatch}.events.dom-mousemove': {
          nextState: 'dragging'
        }
      },
      dragging: {
        '{colorSwatch}.events.dom-mousemove': {
          action: '{that}.events.drag.fire'
        },
        '{colorSwatch}.events.dom-mouseup': {
          nextState: 'idle'
        }
      }
    }
  },
  // the model and listeners define the state machine's
  // internal state and side-effects
  model: {
    offset: {
      x: 0,
      y: 0
    }
  },
  events: {
    dragStart: null,
    drag: null
  },
  listeners: {
    'dragStart.impl': {
      func: 'defs.getOffset',
      args: ['{event}', '{colorSwatch}.model.position'],
      changePath: 'offset'
    },
    'drag.impl': {
      func: 'defs.getNewPosition',
      args: ['{event}', 'offset'],
      changePath: '{colorSwatch}.model.position'
    }
  }
});
```

Figure 5.9: Mixin to extend the color swatch

```
tangler.define('defs.mixinDistributor', {
  // tangler.component is the basic component type
  parentType: 'tangler.component',
  distributeOptions: {
    applyMixinToAllColorSwatches: {
      record: 'defs.colorSwatchMixin',
      // '{root colorSwatch}' refers to all colorSwatch
      // components under the root component.
      // the sub-path .options.components contains
      // declarations of sub-components to be constructed
      target: '{root colorSwatch}.options.components.draggable.type'
    },
  }
});

// add the distributor to the tree
tangler.make('root.colorSwatchMixinDistributor', 'defs.mixinDistributor');
```

Figure 5.10: Option distribution to enact the mixin

from the starting point (in green) upward until a component matching the first part of the selector, here the root (in orange). Then it traverses the tree downwards from there in breadth-first order, collecting all matches to the second part of the selector (in orange). The blue circles are all the components being traversed, the orange one labeled "3" is the one that matches the selector. Figure 5.7.3 illustrates how the mixin consumes events from the swatch, indicated by the selector '.colorSwatch'. A selector containing only a type name searches upward in the tree, checking, in turn, parents and their immediate children.

The next step of the adaptation is to create an entangler to operate the "click-through" behavior precisely when the swatch is above an appropriate component (listing 5.11). Figure 5.7.4 illustrates an entanglement component immediately after creation. Its sub-components are copies-by-reference of the components matched by the co-occurrence engine (drawn with dashed outlines). The template immediately injects the variant event listener via the selector '{that}.swatch.events.activate' (path shown in orange). Figure 5.7.5 shows how the injected listener triggers the activate event.

```
130 tangler.define('defs.
        swatchOverSVGElementEntangler', {
131   // descendants of entangler configure
        the co-occurrence
132   // engine module
133   parentType: 'tangler.entangler',
134   // in co-occurrence conditions, type
        selectors are prefixed
135   // with colons;
136   // 'matches' and 'overlaps' are
        predicate functions defined
137   // in the co-occurrence engine;
138   // 'swatch' and 'target' are variable
        names that will be
139   // assigned to the components that match
140   conditions: [
141     'swatch matches :colorSwatch',
142     'target matches :svgComponent',
143     'swatch overlaps target'
144   ],
145   entanglementTemplate: {
146     // instantiated entanglements will be
        named
147     // swatchOverSVGElement1,
        swatchOverSVGElement2, etc.
148     name: 'swatchOverSVGElement',
149     // instantiated entanglements will be
        of this type
150     type: 'defs.clickThroughSwatch',
151     // the options declaration is used to
        add the co-occurring
152     // components and any additional "
        product" components
153     options: {
154       components: {
155         // '{source}' refers to the
        matched co-occurrence
156         swatch: '{source}.swatch',
157         target: '{source}.target'
158       }
159     }
160   }
161 });
162
163 tangler.define('defs.clickThroughSwatch',
        {
164   // the entanglement type has no special
        behavior,
165   // but is useful for selector matching
166   parentType: 'tangler.entanglement',
167   listeners: {
168     '{that}.swatch.events.activate': {
169       namespace: 'impl',
170       func: 'defs.getFillAndStroke',
171       args: ['{that}.swatch.model'],
172       changePath: '{that}.target.model'
173     }
174   }
175 });
176
177 tangler.make('root.swatchEntangler', 'defs
        .swatchOverSVGElementEntangler');
```

Figure 5.11: Click-through behavior entangler for the color swatch

5.4 Demonstrations

We demonstrate how Tangler enables programmers to extend interactive systems through two example adaptation stories.

Line to Connector

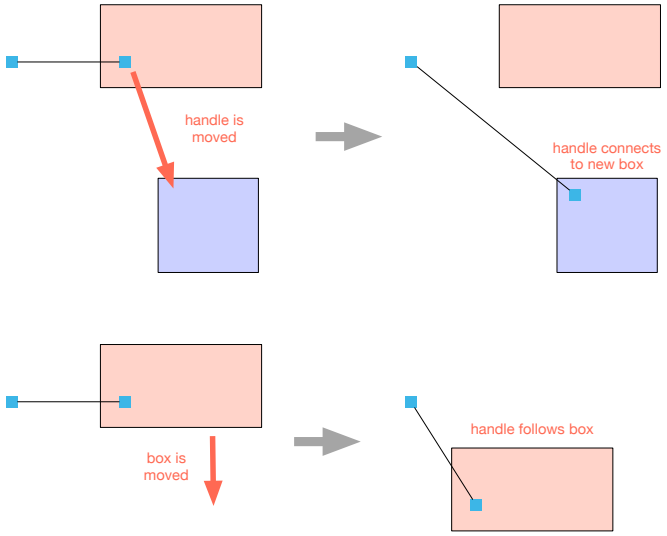

Figure 5.12: The desired behavior associated with the Connector element. Note how the adaptation is not bounded by one object, but adds new behavior both for the handle element and any element it becomes attached to.

The task is to to extend an interactive Line to create a Connector. A Line is a line segment with a draggable handle at each end. A Connector is a line segment whose handles can be attached to arbitrary graphical objects by being dragged onto them (figure 5.12, top). When an attached object (a node) moves, the connected handle moves with it (figure 5.12, bottom). This example is adapted from Lincke et al. (2011).

This adaptation challenges traditional OOP approaches to adaptation because it requires not only specializing the behavior of the line handles to let them be attached and detached, but also contextually specializing the behavior of attached objects, to move any attached handles. It is a useful small example of an interaction that does not cleanly map onto a single self-contained object, but requires integrating the behaviors of multiple distinct objects.

Listing 5.13 shows the initial Line and Handle components. A line would be instantiated by calling tangler.make('root.graphics.line', 'defs.line').

To turn a Line into a Connector, we first define the connector type as an extension of defs.line, which injects an additional sub-component to each of its handles (listing 5.14). This sub-component holds the additional model state and event listeners necessary to manage the

```
tangler.define('defs.line', {
  parentType: 'tangler.viewComponent',
  components: {
    start: {
      type: 'defs.handle'
    },
    end: {
      type: 'defs.handle',
      options: { // override the default position
        model: {
          position: {
            x: 100,
            y: 50
          }
        }
      }
    }
  },
  draw: 'defs.drawLine'
});

tangler.define('defs.handle', {
  parentType: 'tangler.modelComponent',
  model: {
    position: {
      x: 0,
      y: 0
    }
  },
  components: {
    dragging: {
      type: 'defs.dragMixin'
    }
  },
  draw: 'defs.drawHandle'
});

tangler.define('defs.dragMixin', {
  parentType: 'tangler.modelComponent',
  stateMachines: {
    dragMachine: {
      idle: {
        '{viewComponent}.events.dom-mousedown': {
            action: '{that}.events.dragStart.fire',
            nextState: 'dragging'
        }
      },
      dragging: {
        '{viewComponent}.events.dom-mousemove': {
            action: '{that}.events.drag.fire',
            nextState: 'dragging'
        },
        '{viewComponent}.events.dom-mouseup': {
            nextState: 'idle'
        }
      }
    }
  },
  model: {
    offset: {
      x: 0,
      y: 0
    }
  },
  listeners: {
    // store the cursor/object offset when beginning a drag
    'dragStart.impl': {
      func: 'defs.getOffset',
      args: ['{event}', '{viewComponent}.model.position'],
      changePath: 'offset'
    },
    // update the position of the dragged component
    'drag.impl': {
      func: 'defs.getNewPosition',
      args: ['{event}', 'offset'],
      changePath: '{viewComponent}.model.position'
    }
  }
});
```

Figure 5.13: The line component

cycle of attaching and detaching the connector handle from objects. In this case, this behavior can be implemented orthogonally to the existing dragging behavior of line handles.

```
tangler.define('defs.connector', {
  parentType: 'defs.line',
  // put the attachingBehavior component
     on both handles
  distributeOptions: {
    decorateStart: {
      record: 'defs.attachingBehavior',
      target: '{that}.start.options.
     components.connectorHandleManager.
     type'
    },
    decorateEnd: {
      record: 'defs.attachingBehavior',
      target: '{that}.end.options.
     components.connectorHandleManager.
     type'
    }
  }
});

tangler.define('defs.attachingBehavior', {
  parentType: 'tangler.component',
  listeners: {
    '{handle}.events.dom-mouseup': '{that
     }.events.attach.fire',
    '{handle}.events.dom-mousedown': '{
     that}.events.detach.fire'
  },
  events: {
    attach: null,
    detach: null
  },
  invokers: {
    'attach.impl': {
      func: 'defs.
     getViewComponentUnderEvent',
      args: ['{event}'],
      changePath: '{handle}.model.
     attachedNode'
    },
    'detach.impl': {
      value: null,
      changePath: '{handle}.model.
     attachedNode'
    }
  }
});
```

Figure 5.14: The connector component

At this point, connector handles will update their attachedNode and offset fields when dropped on or picked up from graphical elements, but the connector will not follow attached nodes when they move. The second step of the adaptation is to create an entangler that recognizes when handles are attached, instantiates co-occurrences of matching handle-node pairs, and drives the handle-following behavior (listing 5.15).

After providing the new definitions, we must instantiate both the entangler and a connector:

```
tangler.make('root.graphics.connector', 'defs.connector');
tangler.make('root.handleNodeEntangler', 'defs.handleNodeEntangler');
```

It is informative to compare the code involved in this adaptation with the ContextJS example presented by Lincke et al. for the same adaptation (Lincke et al., 2011, pp. 1205-1206). ContextJS uses layers to adapt object behavior. A layer encapsulates adaptations to multiple objects, and can be dynamically activated and deactivated. Lincke et al. create two layers, the first of which changes the behavior of Connector handles, and the second of which changes the behavior of nodes. In their case, handles affected by the Connector layer detect the object they have been dropped on, and apply the Node layer to them. Our adaptation has a similar first step: the connector component adds event listeners

```
tangler.define('defs.handleNodeEntangler', {
  parentType: 'tangler.entangler',
  conditions: [
    'handle matches :connector :handle',
    'node matches :viewComponent',
    'node.id equals handle.attachedNode'
  ],
  entanglementTemplate: {
    name: 'attachedHandleAndNode',
    type: 'defs.connectorEntanglement',
    options: {
      components: {
        handle: '{source}.handle',
        node: '{source}.node'
      }
    }
  }
});

tangler.define('defs.connectorEntanglement', {
  parentType: 'tangler.coOccurrence',
  model: {
    nodePosition: '{that}.node.model.position',
    handlePosition: '{that}.handle.model.position',
    offset: {
      x: 0,
      y: 0
    }
  },
  // in two rules, this component defines the constraint that
  // the handle's position is fixed relative to the node
  modelRelays: {
    updateOffset: {
      target: 'offset',
      transform: {
        x: {
          transformType: 'tangler.sub',
          left: '{that}.model.handlePosition.x',
          right: '{that}.model.nodePosition.x'
        },
        y: {
          transformType: 'tangler.sub',
          left: '{that}.model.handlePosition.y',
          right: '{that}.model.nodePosition.y'
        }
      },
      // always trigger this rule before the next one
      priority: 'before:handleFollowsNode'
    },
    handleFollowsNode: {
      target: 'handlePosition',
      transform: {
        x: {
          transformType: 'tangler.add',
          left: '{that}.model.nodePosition.x',
          right: '{that}.model.offset.x'
        },
        y: {
          transformType: 'tangler.add',
          left: '{that}.model.nodePosition.y',
          right: '{that}.model.offset.y'
        }
      }
    }
  }
});
```

Figure 5.15: The entangler for connectors

to handles that make them store the identifier of the last viewComponent they have been dropped on. The coupling to the second step is different however: rather than imperatively triggering the variation by applying a layer, we reactively trigger the variation by fulfilling the conditions described by defs.handleNodeEntangler.

This different approach showcases a trade-off inherent in our model. On the one hand, it is more computationally expensive to detect the co-occurrence between handles and nodes than simply instantiate the variant behavior in the body of a method that will be called immediately before the behavior is needed (when a handle is dropped on top of an object). On the other hand, our approach is more robust when there are many possible paths to the variant behavior.

Another notable difference is that the ContextJS implementation makes nodes contain handles, because the procedure for moving handles must logically be attached as a listener on nodes. By contrast, in our case, the model relays implementing this behavior (in defs.connectorEntanglement) are attached to the co-occurrences representing each ongoing attachment. We argue that this is a conceptual advantage, especially in more complex integrations, where the co-occurrence provides straightforward access to all involved elements.

One to Two Mice

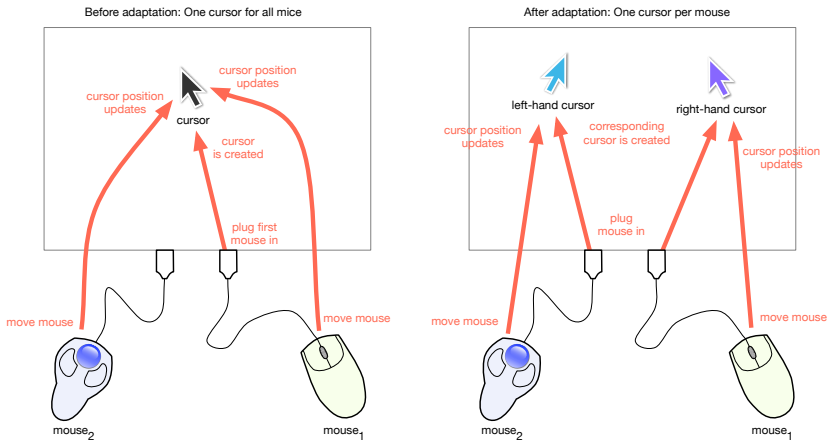

Figure 5.16: The behavior associating mice and cursors before and after the desired adaptation.

The task is to adapt a TANGLER instance that is configured to use a single mouse cursor (as in any modern operating system) so that it instead has uses a different cursor for each mouse plugged into the computer. Before the adaptation, a single cursor will no matter how many mice are plugged in, and the movements of each mouse will be

mapped to that cursor (figure 5.16, left). The adaptation must provide the ability to map each mouse to a different cursor, creating and destroying the cursors as the mice are plugged in and unplugged (figure 5.16, right).

Listing 5.17 shows the code implementing the initial mouse policy. defs.mouseEntanglement defines the cursor, its location, and the means of connecting each mouse to the cursor.

```
tangler.define('defs.cursor', {
  parentType: 'tangler.viewComponent',
  model: {
    position: {
      x: 0,
      y: 0
    },
    radius: '10 px'
  },
  draw: 'defs.drawCursor'
});

tangler.define('defs.mouseEntangler', {
  parentType: 'tangler.entangler',
  conditions: [
    'mouse matches :mouse'
  ],
  entanglementTemplate: {
    name: 'mouseExists',
    type: 'defs.mouseEntanglement',
    options: {
      mouse: '{source}.mouse',
    }
  }
});

tangler.define('defs.mouseEntanglement', {
  parentType: 'tangler.coOccurrence',
  distributeOptions: {
    // create a cursor at the constant location {ui}.cursor
    // '{ui}' resolves to a component under the root component,
    // which contains all elements in the "interface" layer.
    // redundant instances of this distribution have no effect
    createCursor: {
      record: 'defs.cursor',
      target: '{ui}.options.components.cursor.type'
    }
  },
  components: {
    // mirror the created cursor locally
    cursor: '{ui}.cursor',
    // create a transducer connecting
    // the detected mouse and created cursor
    transducer: {
      type: 'defs.mouseCursorTransducer',
      options: {
        components: {
          mouse: '{that}.mouse',
          cursor: '{ui}.cursor'
        }
      }
    }
  }
});

tangler.define('defs.mouseCursorTransducer', {
  parentType: 'tangler.component',
  distributOptions: {
    forwardMouseEvents: {
      record: {
        'device-mousedown': '{coOccurrence}.cursor.events.device-mousedown',
        // ... additional forwarded events omitted ...
      },
      target: '{coOccurrence}.cursor.events'
    }
  },
  modelRelays: {
    // cursor.position = cursor.position + mouse.delta
    // rules using un-invertible transforms work one way
    cursorAccumulatesMouse: {
      target: '{that}.cursor.model.position',
      transform: {
        x: {
          transformType: 'tangler.add',
          left: '{that}.mouse.model.delta.x',
          right: '{that}.cursor.model.position.x'
        },
        y: {
          transformType: 'tangler.add',
          left: '{that}.mouse.model.delta.y',
          right: '{that}.cursor.model.position.y'
        }
      }
    }
  }
});
```

Figure 5.17: The initial mouse policy

The key to this adaptation is to replace it with a component that can dynamically instantiate different cursor configurations depending on the particular mouse that it is created for. To accomplish this, we create a record that maps a mouse identifier, which identifies the hardware

model, to a type defining the appropriate configuration (listing 5.18).

Figure 5.18: The mouse entangler

```
tangler.define('defs.bimanualMouseEntanglement', {
  parentType: 'tangler.coOccurrence',
  mouseToCursorMap: {
    'mouse-1133-49232': 'defs.rightHandCursorConfiguration',
    'mouse-1149-4099': 'defs.leftHandCursorConfiguration',
    // 'default' value is returned for invalid keys
    // this creates a ''catchall'' cursor for additional mice
    'default': 'defs.genericCursorConfiguration'
  },
  components: {
    cursorConfiguration: {
      // rather than provide a static type name here,
      // resolve one dynamically based on the mouse id.
      // the array-form path avoids ambiguous parsing
      type: ['{that}', 'mouseToCursorMap', '{that}.mouse.id'],
      options: {
        components: {
          mouse: '{that}.mouse'
        }
      }
    }
  }
});

tangler.define('defs.rightHandCursorConfiguration', {
  parentType: 'tangler.component',
  distributeOptions: {
    createCursorRemotely: {
      record: 'defs.cursor',
      target: '{ui}.options.components.rightHandCursor.type'
    },
    mirrorCursorLocally: {
      record: '{ui}.rightHandCursor',
      target: '{that}.options.components.cursor'
    }
  },
  components: {
    transducer: {
      type: 'defs.mouseCursorTransducer',
      options: {
        mouse: '{that}.mouse',
        cursor: '{that}.cursor',
      }
    }
  }
});

// ... Other cursor configurations omitted ...
```

The last step is to add a component to the mouse entangler that replaces the old entanglement with the new one (listing 5.19).

Figure 5.19: The mouse distributor

```
tangler.define('defs.bimanualMousePolicyDistributor', {
  parentType: 'tangler.component',
  distributeOptions: {
    setMousePolicy: {
      record: 'defs.bimanualMouseCoOccurrence',
      target: '{mouseEntangler}.entanglementTemplate.type'
    }
  }
});

tangler.make('root.mouseEntangler.options.components.mousePolicy', 'defs.bimanualMousePolicyDistributor');
```

This example illustrates the generality and power of our model. In particular, it demonstrates how entanglers can extend the space of variation of an interactive system in a robust way: the result of this adaptation is a system that provides a meaningful interface whether there are zero or more mice plugged in.

In the above example, defs.rightHandCursorConfiguration applies the same cursor and transducer types to each new mouse as the original setup. In practice it would likely be customized, at a minimum to visually

distinguish the right- and left-hand cursors. This would not have to happen immediately however, as the new cursors may be customized independently by distributing further options to them.

5.5 Discussion

Comparing entanglers and context-oriented programming

Entanglers bear some similarities to previous work in the context-oriented programming (COP) field. Typical COP systems let programmers write layers that encapsulate extensions to several objects. Commonly, layers are applied at the point of a procedure call, and are active throughout the resulting call stack, where they redirect method and field names to context-appropriate values. Some systems have demonstrated other techniques for activating layers. For example, Lincke et al. (2011) demonstrate layer activation applied to particular objects, or with particular containment relationships,i.e. a layer can be active on instances of class A only when they are contained in objects of class B, and not otherwise. Von Löwis et al. (2007) introduce the notion of implicit layer activation, where layers are active while some predicate is true, rather than when a programmer calls an activation method.

Our programming model can be characterized as a COP system with both explicit and implicit layer activation, as well as an extensible language for describing activation contexts. Constructing a component that distributes options to existing components is similar in principle to explicit layer activation, though it is a declarative system extension rather than an imperative one. Co-occurrence descriptions are analogous to layer activation predicates, in that they are automatically matched by the system. They constitute a language for describing dynamically discoverable ensembles of objects. This language can express both object instances and containment relationships, as seen in section 5.4, as well as more novel inter-object relationships, such as visual overlap, seen in section 5.3.

Ramson et al. (2017) find that most COP systems only adapt *passive entities*, i.e. parts of systems that only affect system behavior when they are called into. They argue that implicit layer activation is necessary for adapting active entities that affect system behavior without explicitly receiving control flow. Ramson et al. investigate implicit layer activation implementation strategies, but do not explore applications. Our example in section 5.3 demonstrates exactly the advantage they describe: there is not one known function call or event that must oc-

cur before the co-occurrence between the swatch and target is active, because they may be moved separately.

These congruencies between our model and COP imply that COP may have more to contribute to malleable software. COP research has generally been motivated by programmer-facing concerns such as code maintainability and flexibility. Malleable software may be an avenue for bringing end user concerns into the COP community e.g. by describing malleable use cases as benchmarks to motivate programming tools.

Shifting the line between programming and use

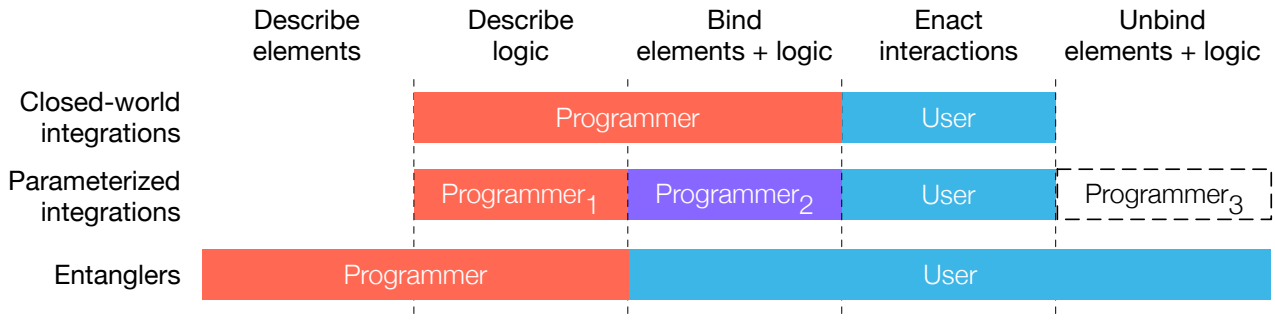

Figure 5.20: A comparison of the distribution of work for various models for programming interactions. The horizontal axis represents the life cycle of an interaction in five phases: describing elements to be integrated, describing the behavior of the interaction, activating the interaction by binding the behavior to concrete system elements, the period of time where the interaction is in effect, and the moment where it is unbound or destroyed. For each kind of interaction, each phase is enacted either by writing and executing code, or by acting on the interface that houses the interaction.

In figure 5.20, I compare three broad categories of interaction in terms of their authorial stories, i.e. how they are created, applied, and removed.

Closed-world interactions are the most common approach to interaction, modeled by containment relationships, event listeners, etc. These interactions are introduced to a system by a programmer describing the new behavior with references to the concrete elements it orchestrates, thus coupling the steps of describing the interaction logic and binding it to particular things. Generally, they are removed or amended by destructive modification, i.e. rewriting source code. They are appropriate for closed-world designs, where the set of elements that will interact is known and stable.

Parameterized interactions are a higher-level response to closed-world interactions. The most basic example is an event listener binding statement encapsulated by a function that takes the elements to be integrated as input parameters. These types of interactions allow for a hand-off, where different programmers may design interactions and instantiate them. OOP mixins, explicitly activated layers, ICon configurations (Dragicevic and Fekete, 2004), OSCAR setups (Newman

et al., 2002), PalCom assemblies (Svensson Fors et al., 2009), and distributing TANGLER components also belong to this category, because they allow for the same decoupling between interaction design and activation. These abstractions have a key difference from a simple interaction wrapped in a function: they can be easily undone, by calling a deactivation method or destroying them.

Finally, entanglers are an even more open-ended form of interaction. They require a more elaborate programming process, as programmers have to design both the pre-conditions for the interaction and its behavior. However, they also remove the need for explicit activation or deactivation steps. Instead, interactions are activated by the underlying system when users bring about the relevant context, and similarly they are deactivated when the user breaks that context.

The figure illustrates how entanglers can help shift the distribution of work between programmers and end users. As I mentioned in the introduction to chapter 4, a key goal of the vision is that software reusability should be a property of the user experience, rather than just the developer experience. More broadly, infrastructures for plurality aim to transform capabilities of programmers into user capabilities without reducing usability, thus enabling diverse user communities to get more value out of their adaptation work.

5.6 Conclusion

I have introduced a programming model for adapting interactions based on co-occurrences and entanglements, and demonstrated the usefulness of the model with our prototype implementation, TANGLER. The key features of our programming model all contribute to making interactions less brittle in the use context of malleable interfaces. The declarative component tree idiom reduces coupling and increases flexibility by letting adaptations target and amend any value in the tree with option distributions and selector queries. Co-occurrence descriptions let adaptations be associated with temporary configurations of components, removing the need to attach adaptations to specific objects or call sites, and the related need for guard code checking for appropriate pre-conditions. Entanglement templates let programmers decide how to name and structure each entanglement, reducing their reliance on the interfaces of integrated components.

TANGLER has several limitations that should be addressed in a more

mature implementation of our model, i.e. to prototype interfaces at the scale of apps or larger systems. I discuss these as directions for future work in the next chapter.

Our programming model shares similar principles with several context-oriented programming (COP) systems. In this perspective, entanglers represent a novel COP abstraction: a declarative contextual adaptation, where the context is bounded by the existence and state of one or several objects, including relational state such as whether two objects visually overlap or not.

5.7 Key Takeaways from Chapter 5

Our programming model based on entanglers makes interactions robust to run-time re-combination of UI elements, and enables ongoing adaptation of those interactions by programmers. Its key features each contribute to these qualities:

- Option distributions targeted with selector queries enable programmers to additively extend components in the whole tree;
- Co-occurrence descriptions enable programmers to create dynamic components representing temporary ensembles of system elements;
- Entanglement templates enable programmers to attach interactions to co-occurrences; and
- The co-occurrence engine creates and destroys interactions in the background as end users re-combine interface elements.

Compared to existing mechanisms for programming interactions, entanglers distribute the process of designing, constructing, enacting, and destroying interactions differently: programmers can design interactions without having to define the concrete elements participating in the interaction, enabling end users to control when and where interactions are in effect. This makes entanglers a suitable low-level idiom for building malleable software.

6

Conclusions

In this thesis I have addressed user adaptation of digital tools in the contemporary software landscape. Adaptation encompasses a wide range of methods employed by individual users and communities to overcome small- and large-scale mismatches between what they need and what their tools provide. These mismatches include failures to accommodate the physical, cognitive, social, and economic context of technology use. As these use contexts are diverse and changing, adaptation work is an ongoing part of use. The costs and outcomes of adaptation are shaped by the material and relational qualities of software, e.g. the people, tools, and techniques that produce, distribute, share, and maintain it. These conditions have shifted significantly in the decades since HCI researchers began developing theoretical and design tools to study and support adaptation work. Computer artifacts have moved out of the office and multiplied into an ecosystem of devices, apps, and documents. A typical user owns, has access to, and uses multiple artifacts produced by and held in common with different communities. I have investigated the methods, goals, and barriers to adaptation work in this context, and argued that they invite us to study and create novel tools for software production. These tools should aim to involve extended networks of users and developers in design and adaptation of their digital tools.

6.1 Thesis contributions

I synthesized the concept of *modes of software production* by drawing in the Marxist concept of modes of production, the ethnographic theory of infrastructures, and the study of how software evolves and mediates work. This theoretical tool foregrounds questions about how the production of software is organized, who has control of which resources, how ownership is distributed, and how these elements of a mode help reproduce themselves. I have applied the mode of software production framework to characterize *the app paradigm*, in which software is produced and distributed as apps, which offer developers increased flexibility through agile methods and cloud infrastructures, but endanger the labor value of users by limiting interoperation and adaptation.

I carried out a case study of a biodiversity research network to understand the contrasting design practices of grassroots communities and software platform owners. In particular, the case study focused on how adaptation work interacts with cross-community collaborations. I found that grassroots communities combine off-the-shelf, tailored, and custom tools to serve their idiosyncratic needs, but that the resulting hybrid systems are cumbersome to use and maintain. By contrast, large-scale software development tools help platform owners create highly usable interfaces, but also skew developers toward centrally managed, one-size-fits-all design.

I proposed that HCI and computer science researchers should create software development tools that resolve this tension between diverse and scalable system designs. I called this research agenda *infrastructures for plurality*, and briefly considered how it could be applied to generate research problems for data sharing and software deployment tools.

I described *malleable software*, a design vision in which interfaces are made up of interface elements that can be developed by different programmers and (re-)combined by end users. I critiqued existing tools for modifying software through walkthroughs of two scenarios of malleable software use cases, and related the problems I found to existing research problems in software engineering and interaction design.

Finally, I designed a programming model to create and adapt interactions for the conditions of malleable software. This model introduces the concept of *entanglers*, which describe the pre-conditions and mechanisms of interactions separately. Entanglers enable programmers to

define and adapt interactions, and enable end users to re-combine interface elements created by different authors. TANGLER is a proof-of-concept implementation of this model, which I used to demonstrate two in-depth adaptation cases. I found that entanglers have several similarities to context-oriented programming mechanisms.

6.2 Directions for future research

I discuss directions for future research that can build on my contributions and address the limitations of my work.

Building an entangler toolkit

TANGLER is a proof of concept for entanglers, but not a sufficient environment for prototyping usable malleable interfaces.

Performance-wise, it is adequately responsive for the size of example we have presented, but requires more careful redesign if it is to scale up to systems with the complexity of real-world applications. In the near term, I aim to create a more fully featured implementation of the programming model. Currently, co-occurrence descriptions that involve predicates comparing very common component types against each other, such as checking for visual overlap between graphical elements, trigger the worst-case performance of the Rete algorithm. This form of co-occurrence is very common in complex GUIs, and may motivate adapting a different matching algorithm. This limitation may also be addressed by applying existing libraries or optimization techniques for graphical interaction. Additionally, the TANGLER component tree is complicated to construct because each component may affect the construction of any other component in the tree through its option distributions. Currently, TANGLER rebuilds the component tree from scratch whenever a new component is introduced, but this is likely to incur significant performance penalties when the tree grows in size. The developers of the Infusion framework that TANGLER is based on are working on an improved construction process, which I hope to take advantage of in future versions of the tool.

My experiments with TANGLER have also opened some subtle design problems for the entangler model. One problem is when entanglements should be destroyed. Currently, TANGLER destroys any entanglements created by an entangler when the triggering co-occurrence ceases to exist. However, this may not be appropriate for some kinds of

interactions, e.g. when a mouse cursor is used to move a scrollbar, the cursor may leave the scrollbar while the mouse key remains depressed without breaking the interaction. It may be viable to let programmers choose how entanglers should destroy entanglements on a case-by-case basis. Another problem is the potential for conflicts and ambiguity among interactions. For example, if button-pressing and dragging interaction and dragging interactions are defined separately, it may not be possible for users to perform these actions separately. These kinds of issues are a common concern of interface toolkits and are typically addressed by mechanisms such as event management (Hudson et al., 2005) or ambiguity resolution (Schwarz et al., 2010). I am currently experimenting with adding these mechanisms to TANGLER as types of entanglements. This should enable programmers to create entanglers that mediate conflicts and ambiguities in existing interactions.

Another pragmatic limitation of entanglers and TANGLER is that they introduce a significant amount of fresh conceptual language. This language has been useful in co-developing the model and implementation with my collaborators (Basman et al., 2018b), but may prevent the wider research community from building on our work. I am interested in developing benchmarks and evaluation methods that operationalize malleability and open authorship. For example, benchmarks may be based on the scenarios from chapter 4. With Antranig Basman, I am considering developing an evaluation method for programming tools based on differential design tasks, i.e. defining a set of related programming exercises that a set of programmers have to solve by sequentially adapting the same program artifact. These tools should help situate our work in the programming literature, and motivate other researchers to contribute their efforts.

Design principles for molding interfaces

I am particularly interested in investigating how non-programming end users may take advantage of entanglers. MacLean et al.'s *Buttons* system (1990) demonstrates the power of adding multiple tailoring techniques with different levels of simplicity and expressiveness to a highly adaptable base system. Such a "gentle slope" to system tailoring can enable collaborative design among end users and developers, as well as reducing the overall need for programming interventions. The discussion in section 5.5 motivates my hypothesis that entanglers are a suitable low-level idiom for empowering users to adapt systems by combining different interface elements at run time. Rather than a gentle slope from use to programming, TANGLER currently exhibits a chasm between a few simple techniques on one side (overlaying in-

terface elements and plugging in devices), and programming on the other.

In the medium term, I am interested in studying and designing tools that can populate this middle ground in tailoring techniques for malleable software. I reviewed several interaction techniques that help users customize interfaces via direct manipulation, programming-by-demonstration, reusable integrations of devices, and visual programming tools in section 4.2 and 5.1. These form a great starting point for this research. The other ongoing work in the ONE project (Beaudouin-Lafon, 2017) will also develop interface concepts and design principles for malleable systems. Indeed interaction instruments and information substrates have been conceptualized from the start as building blocks of interfaces that end users can recombine in the process of use.

Infrastructures for plurality

It was relatively late in my thesis research that I decided to expand my investigation to the larger socio-technical systems around software use and production. Doing this work let me re-evaluate the motivations behind my initial research agenda. I believe that modes of software production and infrastructures for plurality will be useful critical and generative concepts for directing my long-term research efforts. I hope to work on implementing and evaluating tools that support the design work carried out by users and communities on the margins of our software landscape. This will likely involve a transition toward methods such as field studies and co-design.

6.3 Concluding comments

Part of the reason that software so often fails to suit human diversity is that the methods and tools of software production have been developed for the values of industrial-scale engineering. There are many future research contributions in reforming these tools of production. By studying and transforming the extended life cycle of software, rather than just the interface, I believe researchers in HCI and other fields can enable new networks of use and design. I hope that the theoretical and technological tools I have developed will help with these efforts.

Bibliography

Altaweel, I., Good, N., and Hoofnagle, C. J. (2015). Web privacy census. Technology Science.

Anderson, M. (2015). Pew research center. technology device ownership: 2015.

Anderson, M. and Perrin, A. (2017). Pew research center. tech adoption climbs among older adults.

Appert, C. and Beaudouin-Lafon, M. (2006). Swingstates: Adding state machines to the swing toolkit. In Pierce, J., editor, *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, New York, NY, USA. ACM.

Basman, A. (2016). Building software is not (yet) a craft. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group*.

Basman, A. (2019). The naturalist's friend: A case study and blueprint for pluralist data tools and infrastructure. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2019)*.

Basman, A., Church, L., Klokmose, C., and Clark, C. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2016)*.

Basman, A., Clark, C., and Lewis, C. (2015). Harmonious authorship from different representations (work in progress). In *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2015)*.

Basman, A., Lewis, C., and Clark, C. (2018a). The open authorial principle: Supporting networks of authors in creating externalisable de-

signs. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 29–43, New York, NY, USA. ACM.

Basman, A., Tchernavskij, P., Bates, S., and Beaudouin-Lafon, M. (2018b). An anatomy of interaction: Co-occurrences and entanglements. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, pages 188–196, New York, NY, USA. ACM.

Beaudouin-Lafon, M. (2000). Instrumental interaction: An interaction model for designing post-wimp user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '00*, pages 446–453.

Beaudouin-Lafon, M. (2017). Towards unified principles of interaction. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*, CHItaly '17, pages 1:1–1:2, New York, NY, USA. ACM.

Beaulieu, A. (2010). From co-location to co-presence: Shifts in the use of ethnography for the study of knowledge. *Social Studies of Science*, 40(5):453–470.

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA. ACM.

Biddle, R. and Tempero, E. (1998). Evaluating design by reusability.

Bier, E. A. (1992). Embeddedbuttons: Supporting buttons in documents. *ACM Trans. Inf. Syst.*, 10(4):381–407.

Bier, E. A., Stone, M. C., Fishkin, K., Buxtonf, W., and Baudel, T. (1995). A taxonomy of see-through tools. In *Readings in Human–Computer Interaction*, pages 517–523. Elsevier.

Bier, E. A., Stone, M. C., Pier, K., Fishkin, K., Baudel, T., Conway, M., Buxton, W., and DeRose, T. (1994). Toolglass and magic lenses: The see-through interface. In *Conference Companion on Human Factors in Computing Systems*, CHI '94, pages 445–446, New York, NY, USA. ACM.

Blanch, R. and Beaudouin-Lafon, M. (2006). Programming rich interactions using the hierarchical state machine toolkit. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, pages 51–58, New York, NY, USA. ACM.

Bødker, S. (1991). *Through the Interface - a Human Activity Approach to User Interface Design*, volume 16. Lawrence Erlbaum Associates, Hillsdale, NJ.

Bødker, S. and Klokmose, C. N. (2012). Dynamics in artifact ecologies. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, pages 448–457. ACM.

Bødker, S., Korsgaard, H., Lyle, P., and Saad-Sulonen, J. (2016a). Happenstance, strategies and tactics: Intrinsic design in a volunteer-based community. In *Proceedings of the 9th Nordic Conference on Human-Computer Interaction*, NordiCHI '16, pages 10:1–10:10, New York, NY, USA. ACM.

Bødker, S., Korsgaard, H., and Saad-Sulonen, J. (2016b). 'a farmer, a place and at least 20 members': The development of artifact ecologies in volunteer-based communities. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, pages 1142–1156, New York, NY, USA. ACM.

Bødker, S., Lyle, P., and Saad-Sulonen, J. (2017). Untangling the mess of technological artifacts: Investigating community artifact ecologies. In *Proceedings of the 8th International Conference on Communities and Technologies*, C&T '17, pages 246–255, New York, NY, USA. ACM.

Bx (2018). Bx 2018 seventh international workshop on bidirectional transformations. Organized by Kazutaka Matsuda and Jens Weber.

Chen, R. (2019). How to build a plugin system on the web and also sleep well at night.

Chen, X. A. and Li, Y. (2017). Improv: An input framework for improvising cross-device interaction by demonstration. *ACM Trans. Comput.-Hum. Interact.*, 24(2):15:1–15:21.

Christensen, H. B. (2010). *Flexible, reliable software: Using patterns and agile development*. Taylor and Francis (Chapman and Hall/CRC).

Clark, C. and Basman, A. (2017). Tracing a paradigm for externalization: Avatars and the gpii nexus. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 31:1–31:5, New York, NY, USA. ACM.

Clark, C. and Shahi, S. (2018). On continuing creativity. In *Proceedings of the Psychology of Programming Interest Group (PPIG 2018)*.

Clement, A. (1993). Looking for the designers: Transforming the invisible infrastructure of computerised office work. *AI & society*, 7(4):323–344.

Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pages 1–10, New York, NY, USA. ACM.

Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA. ACM.

David, P. A. and Bunn, J. A. (1988). The economics of gateway technologies and network evolution: Lessons from electricity supply history. *Information Economics and Policy*, 3(2):165 – 202.

Dixon, M. and Fogarty, J. (2010). Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1525–1534, New York, NY, USA. ACM.

Doorenbos, R. B. (1995). Production matching for large learning systems. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Dragicevic, P. and Fekete, J. (2002). Icon: input device selection and interaction configuration. In *Companion proceedings of the 15th ACM symposium on User Interface Software & Technology (UIST2), Paris, France*, pages 27–30.

Dragicevic, P. and Fekete, J.-D. (2004). The input configurator toolkit: Towards high input adaptability in interactive applications. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 244–247, New York, NY, USA. ACM.

Eagan, J. R. (2017). Grab 'n' drop: User configurable toolglasses. In Bernhaupt, R., Dalvi, G., Joshi, A., K. Balkrishan, D., O'Neill, J., and Winckler, M., editors, *Human-Computer Interaction – INTERACT 2017*, pages 315–334, Cham. Springer International Publishing.

Eagan, J. R., Beaudouin-Lafon, M., and Mackay, W. E. (2011). Cracking the cocoa nut: User interface programming at runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 225–234, New York, NY, USA. ACM.

Edwards, W. K., Newman, M. W., Sedivy, J. Z., and Smith, T. F. (2009). Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Trans. Comput.-Hum. Interact.*, 16(1):3:1–3:44.

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA. ACM.

Eurostat (2016). Beyond young in europe today - digital world.

Eurostat (2018). Digital econonomy and society statistics - households and individuals.

Fischer, G. (1993). Shared knowledge in cooperative problem-solving systems–integrating adaptive and adaptable components. In *Adaptive User Interfaces*, pages 49–68. Elsevier.

Flanagan, J. C. (1954). The critical incident technique. *Psychological bulletin*, 51(4):327.

Flick, U., Kvale, S., Angrosino, M., Barbour, R., Banks, M., Gibbs, G., and Rapley, T. (2007). *Doing interviews*. SAGE Publications Ltd, London.

Forgy, C. L. (1979). *On the efficient implementation of production systems*. PhD thesis, Carnegie-Mellon University.

Gantt, M. and Nardi, B. A. (1992). Gardeners and gurus: Patterns of cooperation among cad users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 107–117, New York, NY, USA. ACM.

Gasser, L. (1986). The integration of computing and routine work. *ACM Trans. Inf. Syst.*, 4(3):205–225.

Gerson, E. M. and Star, S. L. (1986). Analyzing due process in the workplace. *ACM Transactions on Information Systems (TOIS)*, 4(3):257–270.

Gjerlufsen, T., Klokmose, C. N., Eagan, J., Pillias, C., and Beaudouin-Lafon, M. (2011). Shared substance: Developing flexible multi-surface applications. *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11*, pages 3383–3392.

Gürses, S. and van Hoboken, J. V. J. (2017). Privacy after the agile turn. In polonetsky, J., Tene, O., and Sellinger, E., editors, *Cambridge Handbook of Consumer Privacy*. Cambridge University Press.

Haraty, M., McGrenere, J., and Bunt, A. (2017). Online customization sharing ecosystems: Components, roles, and motivations. In *CSCW*, pages 2359–2371.

Hartmann, B., Doorley, S., and Klemmer, S. R. (2008). Hacking, mashing, gluing: Understanding opportunistic design. *IEEE Pervasive Computing*, 7(3):46–54.

Henderson, A. and Kyng, M. (1995). There's no place like home: Continuing design in use. In *Readings in Human–Computer Interaction*, pages 793–803. Elsevier.

Hudson, S. E., Mankoff, J., and Smith, I. (2005). Extensible input handling in the subarctic toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 381–390, New York, NY, USA. ACM.

Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., Taivalsaari, A., and Mikkonen, T. (2016). A world of active objects for work and play: The first ten years of lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 238–249, New York, NY, USA. ACM.

Jalal, G., Maudet, N., and Mackay, W. E. (2015). Color portraits: From color picking to interacting with color. In *Proceedings of the 2015 CHI Conference on Human Factors in Computing Systems*, CHI '15, pages 4207–4216, New York, NY, USA. ACM.

Jung, H., Stolterman, E., Ryan, W., Thompson, T., and Siegel, M. (2008). Toward a framework for ecologies of artifacts: How are digital artifacts interconnected within a personal life? In *Proceedings of the 5th Nordic Conference on Human-computer Interaction: Building Bridges*, NordiCHI '08, pages 201–210, New York, NY, USA. ACM.

Kaptelinin, V. and Bannon, L. J. (2012). Interaction design beyond the product: Creating technology-enhanced activity spaces. *Human–Computer Interaction*, 27(3):277–309.

Karasti, H. and Blomberg, J. (2018). Studying infrastructuring ethnographically. *Computer Supported Cooperative Work (CSCW)*, 27(2):233–265.

Karasti, H. and Syrjänen, A.-L. (2004). Artful infrastructuring in two cases of community pd. In *Proceedings of the eighth conference on Participatory design: Artful integration: interweaving media, materials and practices-Volume 1*, pages 20–30. ACM.

Kay, A. C. (1984). Computer software. *Scientific American*, 251:53–59.

Kay, A. C. (1993). The early history of smalltalk. *The second ACM SIGPLAN conference on History of programming languages - HOPL-II*.

Kell, S. (2009). The mythical matched modules: Overcoming the tyranny of inflexible software construction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 881–888, New York, NY, USA. ACM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

Klokmose, C. N. and Beaudouin-Lafon, M. (2009). Vigo: Instrumental interaction in multi-surface environments. *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*.

Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W. E., and Beaudouin-Lafon, M. (2015). Webstrates: Shareable dynamic media. *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*, pages 280–290.

Klokmose, C. N. and Zander, P.-O. (2010). Rethinking laboratory notebooks. *Proceedings of COOP 2010*, pages 119–139.

Ko, A. J. and Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA. ACM.

Kuutti, K. (2019). From "interaction" to "transaction" research. Presentation given at the 2019 Common Interactive Objects symposium in Sandbjerg, Denmark.

Lave, J. and Wenger, E. (1991). *Situated learning: Legitimate peripheral participation*. Cambridge university press.

Lialina, O. (2012). Turing complete user. *Contemporary Home Computing*, 14. `contemporary-home-computing.org/turing-complete-user`. Accessed 01/10/2019.

Lieberherr, K., Holland, I., and Riel, A. (1988). Object-oriented programming: An objective sense of style. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '88, pages 323–334, New York, NY, USA. ACM.

Lientz, B. P. and Swanson, E. (1980). *Software maintenance management*. Addison Wesley.

Lincke, J., Appeltauer, M., Steinert, B., and Hirschfeld, R. (2011). An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76(12):1194–1209.

Mackay, W. E. (1990a). Patterns of sharing customizable software. *Proceedings of the 1990 ACM conference on Computer-supported cooperative work - CSCW '90*.

Mackay, W. E. (1990b). *Users and customizable software: A co-adaptive phenomenon*. PhD thesis, Massachusetts Instititute of Technology.

Mackay, W. E. (1991). Triggers and barriers to customizing software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 153–160, New York, NY, USA. ACM.

Mackay, W. E. (2002). Which interaction technique works when?: Floating palettes, marking menus and toolglasses support different task strategies. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, pages 203–208, New York, NY, USA. ACM.

Mackay, W. E. and Fayard, A.-L. (1997). Hci, natural science and design: a framework for triangulation across disciplines. In *Symposium on Designing Interactive Systems: Proceedings of the 2nd conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 223–234.

MacLean, A., Carter, K., Lövstrand, L., and Moran, T. (1990). User-tailorable systems: Pressing the issues with buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 175–182, New York, NY, USA. ACM.

Magnaudet, M., Chatty, S., Conversy, S., Leriche, S., Picard, C., and Prun, D. (2018). Djnn/smala: A conceptual framework and a language for interaction-oriented programming. *Proc. ACM Hum.-Comput. Interact.*, 2(EICS):12:1–12:27.

Maloney, J. H. and Smith, R. B. (1995). Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28, New York, NY, USA. ACM.

Marx, K. (2005). *Grundrisse: Foundations of the critique of political economy*. Penguin UK.

Maudet, N., Jalal, G., Tchernavskij, P., Beaudouin-Lafon, M., and Mackay, W. E. (2017). Beyond grids: Interactive graphical substrates to structure digital layout. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5053–5064, New York, NY, USA. ACM.

McGrenere, J. (2002). *The design and evaluation of multiple interfaces: A solution for complex software*. PhD thesis, University of Toronto Canada.

Myers, B. A. (1990). Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.*, 12(2):143–177.

Myers, B. A. (1991). Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, pages 211–220, New York, NY, USA. ACM.

Nardi, B. A., editor (1996). *Context and consciousness: activity theory and human-computer interaction*. MIT Press.

Nardi, B. A. and Miller, J. R. (1991). Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34(2):161–184.

Nelson, T. H. (2012). *Transclusion: Fixing electronic literature*. GoogleTalksArchive, YouTube. `youtube.com/watch?v=ohiKTVVtDJA`. Accessed 01/03/2016.

Newman, M. W., Elliott, A., and Smith, T. F. (2008). Providing an integrated user experience of networked media, devices, and services through end-user composition. In *International Conference on Pervasive Computing*, pages 213–227. Springer.

Newman, M. W., Sedivy, J. Z., Neuwirth, C. M., Edwards, W. K., Hong, J. I., Izadi, S., Marcelo, K., and Smith, T. F. (2002). Designing for serendipity: Supporting end-user configuration of ubiquitous computing environments. In *Proceedings of the 4th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, DIS '02, pages 147–156, New York, NY, USA. ACM.

Nielsen, M. N. and Bødker, S. (2004). Desktop computing. *Australian Journal of Information Systems, special*, pages 88–101.

Nosek, J. T. and Palvia, P. (1990). Software maintenance management: changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3):157–174.

Nouwens, M. and Klokmose, C. N. (2018). The application and its consequences for non-standard knowledge work. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 399:1–399:12, New York, NY, USA. ACM.

Olsen, Jr., D. R., Hudson, S. E., Verratti, T., Heiner, J. M., and Phelps, M. (1999). Implementing interface attachments based on surface representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 191–198, New York, NY, USA. ACM.

Olsen Jr, D. R. (2007). Evaluating user interface systems research. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 251–258. ACM.

Parnas, D. L. (1971). Information distribution aspects of design methodology. *Information Processing*, pages 339–344.

Paternò, F. and Wulf, V. (2017). *New Perspectives in End-User Development*. Springer.

Pollock, N. and Williams, R. (2010). E-infrastructures: How do we know and understand them? strategic ethnography and the biography of artefacts. *Computer Supported Cooperative Work (CSCW)*, 19(6):521–556.

Ramson, S., Lincke, J., and Hirschfeld, R. (2017). The declarative nature of implicit layer activation. In *Proceedings of the 9th International Workshop on Context-Oriented Programming*, COP '17, pages 7–16, New York, NY, USA. ACM.

Schmidt, K. and Bannon, L. (1992). Taking cscw seriously. *Computer Supported Cooperative Work (CSCW)*, 1(1-2):7–40.

Schwarz, J., Hudson, S., Mankoff, J., and Wilson, A. D. (2010). A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 47–56, New York, NY, USA. ACM.

Shneiderman, B. (1997). Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of the 2Nd International Conference on Intelligent User Interfaces*, IUI '97, pages 33–39, New York, NY, USA. ACM.

Star, S. L. (1999). The ethnography of infrastructure. *American behavioral scientist*, 43(3):377–391.

Star, S. L. and Bowker, G. C. (2002). *How to Infrastructure*, pages 151–162. SAGE Publications, Ltd, Thousands Oaks, CA, USA.

Star, S. L. and Ruhleder, K. (1996). Steps toward an ecology of infrastructure: Design and access for large information spaces. *Information systems research*, 7(1):111–134.

Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., and Tatar, D. (1987). Wysiwis revised: Early experiences with multiuser interfaces. *ACM Trans. Inf. Syst.*, 5(2):147–167.

Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. (2006). User interface façades: Towards fully adaptable user interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 309–318, New York, NY, USA. ACM.

Suchman, L. (2002). Located accountabilities in technology production. *Scandinavian journal of information systems*, 14(2):7.

Sutherland, I. E. (1963). Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA. ACM.

Svensson Fors, D., Magnusson, B., Gestegård Robertz, S., Hedin, G., and Nilsson-Nyman, E. (2009). Ad-hoc composition of pervasive services in the palcom architecture. In *Proceedings of the 2009 International Conference on Pervasive Services*, ICPS '09, pages 83–92, New York, NY, USA. ACM.

Trigg, R. H. and Bødker, S. (1994). From implementation to design: Tailoring and the emergence of systematization in cscw. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 45–54, New York, NY, USA. ACM.

Trigg, R. H., Moran, T. P., and Halasz, F. G. (1987). Adaptability and tailorability in notecards. In *Human–Computer Interaction–INTERACT'87*, pages 723–728. Elsevier.

Ungar, D. and Smith, R. (2013). The thing on the screen is supposed to be the actual thing. In *International Workshop on Live Programming at the International Conference on Software Engineering*.

Ur, B., Pak Yong Ho, M., Brawner, S., Lee, J., Mennicken, S., Picard, N., Schulze, D., and Littman, M. L. (2016). Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3227–3231, New York, NY, USA. ACM.

Vitale, F., McGrenere, J., Tabard, A., Beaudouin-Lafon, M., and Mackay, W. E. (2017). High costs and small benefits: A field study of how users experience operating system upgrades. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4242–4253, New York, NY, USA. ACM.

Von Löwis, M., Denker, M., and Nierstrasz, O. (2007). Context-oriented programming: beyond layers. In *International Conference on Dynamic Languages (ICDL)*, pages 143–156.

Wulf, V., Pipek, V., and Won, M. (2008). Component-based tailorability: Enabling highly flexible software applications. *International Journal of Human-Computer Studies*, 66(1):1–22.

Zhou, J. (2003). A history of web portals and their development in libraries. *Information Technology and Libraries*, 22(3):119–128.

Résumé Détaillé

Les besoins des utilisateurs en matière de fonctionnalités et d'interfaces logicielles sont variés et changeants. Mon objectif est de permettre aux utilisateurs eux-mêmes de facilement modifier ou faire modifier leur logiciel en fonction de l'évolution de leurs besoins. Toutefois, à mon avis, les approches actuelles ne traitent pas cette question de façon adéquate: L'ingénierie logicielle favorise la flexibilité du code mais, dans la pratique, cela n'aide pas les utilisateurs finaux à apporter des changements à leurs logiciels. Les systèmes permettant à l'utilisateur de programmer en direct ("live programming") ou de modifier le code du logiciel ("end-user programming") permettent aux utilisateurs de personnaliser les interfaces de leur logiciel en accédant et modifiant le code source. J'adopte une approche différente, qui cherche à maximiser les modifications qui peuvent être faites à travers des interactions habituelles, par exemple la manipulation directe d'éléments d'interface. J'appelle cette approche la malléabilité logicielle. Pour comprendre les besoins des utilisateurs et les obstacles à la modification des logiciels interactifs, j'étudie comment les logiciels actuels sont produits, maintenus, adoptés et appropriés dans un réseau de communautés travaillant avec des données sur la biodiversité. J'ai interrogé certains intervenants principaux de et autours de cette communauté, et j'ai cartographié leur écosystème de technologies composé par exemple de code source, fichiers partagés, applications, et sites Internet. J'analyse ces données à l'aide de concepts issus du travail coopératif assisté par ordinateur (computer-supported cooperative work), des études scientifiques et technologiques, du génie logiciel et de l'économie marxiste. Je montre que le mode de production des logiciels, c'est-à-dire les technologies et les modèles économiques qui les produisent, est biaisé en faveur de systèmes centralisés et uniformisés. Cette tendance est en conflit avec les besoins des petites communautés d'utilisateurs, comme celle que j'etudie, qui n'ont pas les moyens de transformer et d'intégrer leurs outils numériques en fonction de leurs

besoins locaux. Lorsque ces communautés s'approprient leurs outils, elles doivent payer des coûts de main-d'œuvre continus pour maintenir et partager ces outils. Par contre, les développeurs de logiciels professionnels ont des infrastructures spécialisée permettant d'éviter ou de réduire ces coûts. Cela m'amène à proposer un programme de recherche interdisciplinaire à long terme pour repenser les outils de développement logiciel afin de créer des infrastructures pour la pluralité (de platform, de document, de base de données entre autre). Ces outils pourraient aider de multiples communautés à collaborer sans les forcer à adopter des interfaces ou représentations de données identiques. Le logiciel malléable est un exemple d' une telle infrastructure, dans laquelle les systèmes interactifs sont des constellations dynamiques d'interfaces, de dispositifs et de programmes construits au moment de leur utilisation. Ma contribution technologique est de recréer des mécanismes de programmation pour concevoir des comportements interactifs. Je généralise les structures de contrôle existantes pour l'interaction en ce que j'appelle des intrications ("entanglements"). J'élabore une structure de contrôle d'ordre supérieur, les intricateurs ("entanglers"), qui produisent ces intrications lorsque des conditions préalables particulières sont remplies. Ces conditions préalables sont appelées co-occurrences. Les intricateurs organisent l'assemblage des interactions dynamiquement en fonction des besoins des composants du système. Je développe ces mécanismes dans Tangler, un prototype d'environnement pour la construction de logiciels interactifs malléables. Je démontre comment Tangler supporte la malléabilité à travers un ensemble de cas d'étude illustrant comment les utilisateurs peuvent modifier les systèmes par eux-mêmes ou avec l'aide d'un programmeur. Cette thèse est un premier pas vers un paradigme de programmation et de conception de logiciels malléables capables de s'adapter à la diversité des usages et des utilisateurs. Ce paradigme émergent nécessite de nouvelles technologies dans des domaines qui sont généralement considérés comme hors de la portée de l'IHM, tels que les bases de données et les systèmes de gestion de versions, mais nécessite également de nouveaux modèles pour la production et la maintenance collectives des logiciels.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Concevoir et programmer des logiciels malléables

Mots clés : Interaction Homme-Machine, Modélisation Conceptuelle, Ingénierie des Systèmes Interactifs

Résumé : Les besoins des utilisateurs en matière de fonctionnalités et d'interfaces logicielles sont variés et changeants. Mon objectif est de permettre aux utilisateurs eux-mêmes de facilement modifier ou faire modifier leur logiciel en fonction de l'évolution de leurs besoins. Toutefois, à mon avis, les approches actuelles ne traitent pas cette question de façon adéquate: L'ingénierie logicielle favorise la flexibilité du code mais, dans la pratique, cela n'aide pas les utilisateurs finaux à apporter des changements à leurs logiciels. Les systèmes permettant à l'utilisateur de programmer en direct ("live programming") ou de modifier le code du logiciel ("end-user programming") permettent aux utilisateurs de personnaliser les interfaces de leur logiciel en accédant et modifiant le code source. J'adopte une approche différente, qui cherche à maximiser les modifications qui peuvent être faites à travers des interactions habituelles, par exemple la manipulation directe d'éléments d'interface. J'appelle cette approche la malléabilité logicielle. Pour comprendre les besoins des utilisateurs et les obstacles à la modification des logiciels interactifs, j'étudie comment les logiciels actuels sont produits, maintenus, adoptés et appropriés dans un réseau de communautés travaillant avec des données sur la biodiversité. Je montre que le mode de production des logiciels, c'est-à-dire les technologies et les modèles économiques qui les produisent, est biaisé en faveur de systèmes centralisés et uniformisés. Cela m'amène à proposer un programme de recherche interdisciplinaire à long terme pour repenser les outils de développement logiciel afin de créer des infrastructures pour la pluralité. Ces outils peuvent aider de multiples communautés à collaborer sans les forcer à adopter des interfaces ou représentations de données identiques. Le logiciel malléable représente une telle infrastructure, dans laquelle les systèmes interactifs sont des constellations dynamiques d'interfaces, de dispositifs et de programmes construits au moment de leur utilisation. Ma contribution technologique est de recréer des mécanismes de programmation pour concevoir des comportements interactifs. Je généralise les structures de contrôle existantes pour l'interaction en ce que j'appelle des intrications ("entanglements"). J'élabore une structure de contrôle d'ordre supérieur, les intricateurs ("entanglers"), qui produisent ces intrications lorsque des conditions préalables particulières sont remplies. Ces conditions préalables sont appelées co-occurrences. Les intricateurs organisent l'assemblage des interactions dynamiquement en fonction des besoins des composants du système. Je développe ces mécanismes dans Tangler, un prototype d'environnement pour la construction de logiciels interactifs malléables. Je démontre comment Tangler supporte la malléabilité à travers un ensemble de cas d'étude illustrant comment les utilisateurs peuvent modifier les systèmes par eux-mêmes ou avec l'aide d'un programmeur. Cette thèse est un premier pas vers un paradigme de programmation et de conception de logiciels malléables capables de s'adapter à la diversité des usages et des utilisateurs.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Designing and Programming Malleable Software

Keywords : Human-Computer Interaction, Conceptual Modeling, Engineering of Interactive Systems

Abstract : User needs for software features and interfaces are diverse and changing, motivating the goal of making it as easy as possible for users themselves to change software, or to have it changed on their behalf in response to their developing needs. However, in my opinion, current approaches do not address this issue adequately: software engineering promotes flexible code, but in practice this does not help end-users effect change in their software. End-user and live programming systems help users customize their interfaces by accessing and modifying the underlying source code. I take a different approach, seeking to maximize the kinds of modifications that can take place through regular interactions, e.g. direct manipulation of interface elements. I call this approach malleable software. To understand contemporary needs for and barriers to modifying software, I study how it is produced, maintained, adopted, and appropriated in a network of communities working with biodiversity data. I find that the mode of software production, i.e. the technologies and economic relations that produce software, is biased towards centralized, one-size-fits-all systems. This leads me to propose a long-term, interdisciplinary research program in reforming the tools of software development to create infrastructures for plurality. These tools should help multiple communities collaborate without forcing them to consolidate around identical interfaces or data representations. Malleable software is one such infrastructure, in which interactive systems are dynamic constellations of interfaces, devices, and programs assembled at the site of use. My technological contribution is a reconstruction of the programming mechanisms used to create interactive behavior. I generalize existing control structures for interaction as entanglements, and develop a higher-order control structure, entanglers, which produces entanglements when particular pre-conditions, called co-occurrences, are met. Entanglers cause interactions to be assembled dynamically as system components come and go. I develop these mechanisms in Tangler, a prototype environment for building malleable interactive software. I demonstrate how Tangler supports malleability through a set of benchmark cases illustrating how users can modify systems by themselves or with programmer assistance. This thesis is an early step towards a paradigm for programming and designing malleable software that can keep up with human diversity.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France