In Christoph Meinel and Larry Leifer (eds.)
Design Thinking Research: Looking Further: Design Thinking Beyond Solution-Fixation (pages 229-247)
Springer 2018 (doi:10.1007/978-3-319-97082-0_12)

Towards Exploratory Software Design Environments for the Multi-disciplinary Team

Patrick Rein, Marcel Taeumel, and Robert Hirschfeld

Software Architecture Group, Hasso Plattner Institute, University of Potsdam, 14482 Potsdam, Germany

Abstract The creation of a new software system can be a wicked problem. Consequently, it is important for such projects to have a collaborating team of experts from multiple disciplines. While agile development processes foster such a collaboration on the social level, the tools used by individual experts still prevent the team members from seeing the overall result of their collective modifications upon the resulting system. Roles in the process, such as content designers and user experience designers, only see the impact of their changes on their artifacts. Based on the concept of exploratory programming environments, we propose a new perspective on the environments used in software development, called *exploratory software design environments*. We describe the properties of such an environment and illustrate the perspective with existing related tools and environments.

1 Introduction

Software development has the properties of a wicked problem: Requirements might only become apparent after an interim solution was proposed and software is never done as the intended real-world use cases for the software constantly change [Rittel and Webber 1973, Conklin 2006, DeGrace and Stahl 1990]. Further, the creators of a software system have to account for a variety of properties such as technical stability and maintainability, usability of the user interface, correctness of the domain model, and actual usefulness to the users. Consequently, software development can benefit from insights of the Design Thinking methodology, here the consideration of multiple viewpoints for solving such wicked problems [DeGrace and Stahl 1990, Beck 2000].

In order to create an appropriate solution, a multi-disciplinary team has to closely collaborate, as only then the multiple perspectives of the participants can actually contribute to the design. In such close collaboration, team members are not only interested in finishing their individual tasks but *continuously assess the impact of* *their own contributions* on the overall design and comments on any other contribution if they think it necessary. A commonly described factor for creating such a team culture is the creation of a common purpose within the team.

Software development can benefit from teams of experts that incorporate multidisciplinary knowledge. The variety of properties of a software system makes it essential for the design process that a variety of people with different backgrounds are involved in the creation of the software, such as back-end developers, user interface designers, the actual users, and experts of the application domain. The common purpose of such software development teams, ideally, is the collective and continuous evolution of a system, which brings value to its users. Agile processes are based on the notion of a team sharing a common purpose. These processes try to support the team culture through appropriate techniques. For example, Extreme Programming (XP) lists "The Whole XP Team" as one of its practices and describes it as "... they [the team] were roped together. Walking abreast, they could make more progress than if any one group tried to force to others to follow." [Beck 2000]

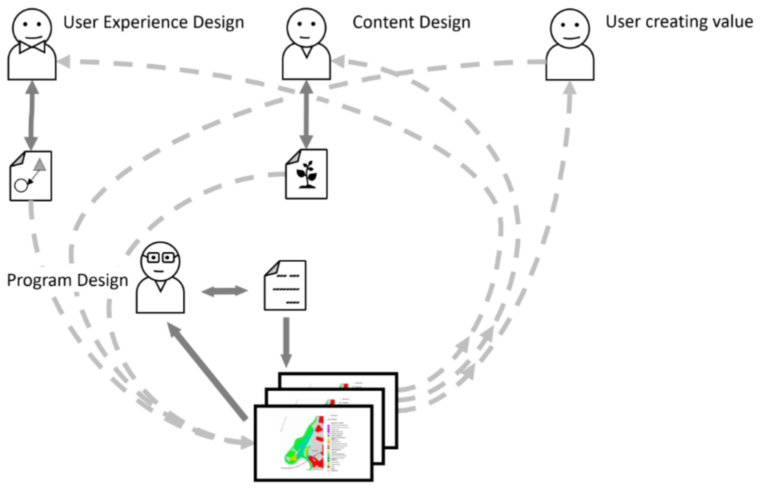


Fig. 1. The current workflow in software development. Most team members work on their type of artifacts (bold arrows). They only get feedback on the impact of their changes on the system at a later point in time after submitting their artifacts to the program designers (dashed lines).

While such development practices aim to support a culture of working together on a single system, this impression is often not reflected in the software tools used by individual participants. All experts operate their own tools, creating an output which is only later combined into the running system. For example, technical writers are often passed a file with a long list of placeholders which they should, for example, translate into full-length labels. The effects of changes to the text might only become visible to the other team members much later in the process and only in case they actually run the new system version. Consequently, even while an agile development process might aim at collaboration on one system, the tools only allow for cooperation almost resembling a *software factory* with single workstations (see Figure 1).

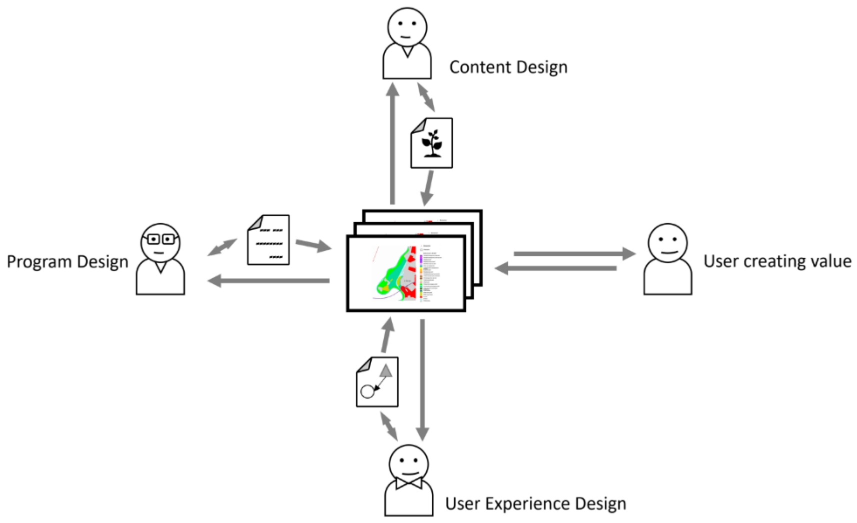


Fig. 2. The ideal workflow in exploratory software design environments for the whole team. Every team member can work on their artifacts but also get direct feedback of their modifications to the system. Further, they can see how their modification interacts with modifications of others.

Instead, we should aim for a *software workshop* in which all participants of the process work together on the actual, running system (see Figure 2). In such a workshop, the technical writer from above would change the labels while the software is running and others would shortly afterwards see the changed labels, too. Whenever someone applies a change to the system, the effect should become visible to the other team members in short time. This facilitates a sense of working together on one system and make collaboration more likely.

In this article, we illustrate the factors impeding collaboration during the design of software in traditional environments. Further, we show how so called *exploratory programming environments* can serve as a foundation for a software workshop in which people with different roles can collaborate directly on one system. We do so by describing the properties of exploratory programming environments and illustrate these properties with two exemplary programming systems. We then generalize these properties to properties for *exploratory software design environments* which provide an exploratory workflow for all participants of the process. To show how such an environment might work we further describe a number of exemplary tools and environments that implement characteristic aspects.

2. From Cooperation to Collaboration in Software Development

Agile processes such as Extreme Programming (XP) share a number of principles and values with the design thinking methodology. Both are iterative in nature, make creating value for the user a primary goal, and emphasize self-sufficient and multi-disciplinary teams. To actually leverage the different viewpoints of a multi-disciplinary team, team members have to collaborate beyond mere cooperation. As software development already entails particular tasks, a multi-disciplinary software development team has a set of artifacts and tools, which correspond to typical roles: content creation, user experience design, program design, and user.

2.1 Design Thinking, Wicked Problems, and Agile Processes

Wicked problems are problems which have "no definitive formulation" due to requirements which are incomplete from the start or might change during the design of a solution. Traditional examples of wicked problems are social problems such as drug abuse and homelessness. In a more general way, they have been defined through a set of six characteristics [Conklin 2006] (derived from a larger catalogue of eleven characteristics [Rittel and Webber 1973]):

1. The problem is not understood until after the formulation of a solution.
2. Wicked problems have no stopping rule.
3. Solutions to wicked problems are not right or wrong.
4. Every wicked problem is essentially novel and unique.
5. Every solution to a wicked problem is a 'one shot operation.'
6. Wicked problems have no given alternative solutions.

The design thinking process is a suitable for approaching wicked problems [Buchanan 1992]. For example, the iterations in the design process allow team members to refine their understanding of the problem after each iteration (characteristic 1). Further, the novel and unique nature of the problem is covered by techniques for ideation to support the team in creating new and fitting solutions (characteristic 4). Even early design thinking methodologies already focused on similar types of problems [Arnold 1956, Arnold 1959/2016]. Further, wicked problems consist of interdependencies of various individual factors. Each factor might only be understood in terms of a particular field, such as sociology, art, and mechanical engineering. Thus, design teams should ideally consist of experts from a variety of fields. Due to their different backgrounds, each team member has a different viewpoint onto the original problem and can hence determine sub-problems related to their field.

The design of software systems is also considered a wicked problem [DeGrace and Stahl 1990]. A major aspect which makes software development wicked is that the actual requirements for a software system are only understood after parts of the software have been build and are in use. Often, users require an intermediate state of the software to determine what they actually need. Further, software also does not have a "stopping rule" (characteristic 2). As the context of the use of software constantly changes, software has to adapt accordingly. This is summarized by the saying in software industry that "software is never done".

Agile processes acknowledge the wicked nature of software development. One of the principles of Extreme Programming (XP) is, for example, "embrace change", meaning that development should happen in small iterations to get feedback from users in time. Based on this feedback, the system can directly be adapted to fit the new requirements best [Beck 2000]. In this respect, every iteration of an XP team is a design iteration. At the beginning, needs and wishes from the users are collected. The team then works out a solution for these challenges and produces a small increment in the features of the system. This increment serves as a prototype which is directly evaluated with the users by incorporating it into the running software. Observations and feedback from the users are directly used in the next iteration. Also, software development covers more than the mere production of source code. Software development also covers activities such as user experience design, interface design, and content creation by domain experts. Consequently, XP also emphasizes a "whole team" and collaboration becomes paramount in the process. Every activity or aspect of the software should be covered by someone in the multi-disciplinary team.

2.2 From Cooperation to Collaboration

For multi-disciplinary teams to work they have to *collaborate* on solving the problem and not only *cooperate*. Although both of the words "cooperation" and "collaboration" generally refer to working together, the style of working together they describe differs.

Cooperation is defined by Meriam Webster as "to act or work with another or others; act together or in compliance". The important part here is that the team members merely *act* together. This does not imply a shared goal or active support for each other. Individuals who cooperate have some overlap of their goals but the individual goals dominate. An example for cooperation are bureaucratic organizations. The group of people who make up the organization cooperate by each fulfilling their individual tasks and thereby providing the service of the organization.

Collaboration is a special form of cooperation defined by Meriam Webster as "to work jointly with others or together especially in an intellectual endeavor". People who collaborate closely work with the others in order to achieve a common goal. Design teams typically collaborate as they discuss ideas in the group together. A major requirement for a team to closely collaborate is that all team members have a shared understanding of the common purpose of the team. Further, every team member has to assess the complete situation continuously and put it relation to the strategic goal of the team, similar to the way each soccer player has to continuously monitor the complete soccer field and not only concentrate on their "patch of grass." [McChrystal 2015]

In his book "Team of Teams", Stanley McChrystal does not mention the two terms, cooperation and collaboration, explicitly but illustrates the two styles of working together through a comparison of traditional command structures in the military and team-based structures in a variety of domains [McChrystal 2015]. The author describes command structures as a way of cooperation "[...] in a command, the leader breaks endeavors down into separate tasks and hands them out. The recipients of instructions do not need to know their counterparts, they only need to listen to their boss. In a command, the connections that matter are vertical ties." The author states, that this cooperation in command structures was efficient but rigid which according was not a good fit for modern challenges. The context and requirements for modern challenges change too quickly for any pre-determined plan to be applicable. Instead, organizations should focus on small empowered teams whose members collaborate working towards a common goal. Again, one of the characteristics of such teams is that "team members tackling complex environments must all grasp the team's situation and overarching purpose [...] They must be collectively responsible for the team's success and understand everything that responsibility entails."

The design of a software system depends on such collaboration between experts from multiple disciplines. However, the fact that these team members are experts of their own discipline and masters of their own tools makes it very easy to stick to their “patch of grass”.

2.3 The Whole Team: Multiple Disciplines for Multiple Perspectives

Software development benefits from a pre-defined type of artifact to be produced: the software system. There are particular roles which are relevant to the software design process [Beck 2000], such as testers, interface designers, programmers, technical writers, and managers. Depending on the type of system to be created, different roles might be more active in the process than others. For example, when developing a computer game artists can make up more than half of the team. Similarly, when working on a software tool for a very particular domain, domain experts might outnumber programmers (for example, the biochemistry software tool company Synthace lists two biochemistry scientists as technical leads and only one software engineer[1]). Thus, it is beneficial for the quality of working together, to enable all these roles to participate equally during the software design.

Further, these roles are only approximate groupings of activities. For example, in XP "roles on a mature XP team aren't fixed and rigid. The goal is to have everyone

[1] https://web.archive.org/web/20171205131307/https://synthace.com/who-we-are/ accessed on 5th of December 2017

contribute the best he has to offer to the team's success." [Beck 2000] We summarize these roles into the four categories:

- User
- User experience designer
- Content designer
- Program designer

We want to illustrate their typical contributions, their tools, and the artifacts they create in an iteration during the development process.

Users provide new requirements and general use cases for the software. They might have thought them out in great details already or only have a vague idea. Generally, users are the ones generating value with the software system produced. They are affected by all the decisions of the other roles and at the same time provide the requirements, use cases, and the overall purpose of the software to be produced. Besides the direct users, we also count customers and domain experts into this role. In consumer software they often interact with the system design team through issue tracking software. In more specialized software they might be able to talk in person to the design team and may even be able to join them during an iteration.

Early in the process *user experience designers* might produce first paper-based sketches of user interfaces to check with users whether this is what they might need. Additionally, they might create storyboards to document a workflow users want to have supported in the system. In general, they determine the actual interactions and feedback mechanisms of the system to make the program useful to the users. They take care of the intricacies of single user interfaces as well as the efficiency complete workflows throughout the system. Activities of this role might also be subsumed under the terms "usability engineer" or "user interface engineer". They work with a variety of tools starting with pencil and paper. For visual designs, user experience designers might use graphical editors in which they create screenshots of the future interface. They might also use user interface builder tools in which they can already define the actual user interface in a graphical manner.

Content designers generally create the texts, graphics, or pre-loaded data and examples used throughout a system. The particular type of output depends upon the domain of the system. Being artists, they create texts and graphics and have to take care of aspects such as a consistent aesthetic appearance or fitting the content to the culture of the system's user. Correspondingly, the tools used also depend on the domain of the system. Either way, most of these tools are specialized to the content format, such as the Adobe Photoshop[2] graphics editor for graphical content or the Qt Linguist[3] for translation tables. Depending on the system the content designers might also be domain experts contributing domain specific knowledge to the system, such as mathematical formulas or business rules.

[2] https://web.archive.org/web/20171205125120/http://www.adobe.com/de/products/photoshop.html accessed on 5th of December 2017

[3] https://web.archive.org/web/20171205125218/http://doc.qt.io/qt-5/linguist-translators.html accessed on 5th of December 2017

Program designers create and maintain the technical side of a software system. This covers the design of the overall system architecture as well as fine-granular decision such as the names used in the source code. Further, as test engineers they might also write automated tests for checking whether the system behaves as expected, or as tool engineers, they might create tools for making the overall design of the system easier for all roles. When working on a new feature, progam designers add, edit, and remove source code. This changes to the source code are often done in a so-called integrated development environments (IDEs) which provides a set of integrated development tools in one environment. When working on the overall structure, program designers often use graphical modelling tools which allow them to draw diagrams representing the system structure.

2.4 The Impact of Tools on Cooperation and Collaboration

Agile processes, such as XP, try to tackle the wickedness of software development similar to the way design thinking methodologies tackle wicked problems. Collaboration of multi-disciplinary teams is a key component of Extreme Programming. Thus, XP lists a number of principles and practices to foster this collaboration in social interactions.
However, when it comes to actually working together on the system to be created, the software tools used by team members do not support close collaboration. Every team member uses their specialized tool set to produce artifacts which are particular to their activity. Regarding the concrete artifact to be produced, team members might get feedback in a short amount of time, for example a content designer creating a new icon can see the icon directly in the graphics editor. However, the impression of the icon in the running system might only become available much later when the files representing the icon are merged into the software system. This is similar to the way production in a factory works: Individual workers work on optimized stations with their specialized tool. The resulting end product might never be visible to them. Both share the property that there is a long delay between a new artifact produced and a visible change in the resulting system might span hours or days.
Such a long delay between one's modifications and an actual change in the software to be created does hinder individual team members to assess the overall state of the software and their impact on it. They work on their local view of the system for an extended period of time. As a consequence, interdependencies between modifications, positive as well as negative ones, can only be detected late in the process. For example, translators have to wait for the merge of their translation tables only to find out that a translated text was too long because the interface designer changed the width of some buttons simultaneously. This scenario that translators find such problems is further based on the unlikely assumption that translators actually do review their translated texts in the user interface at a later point in time. Further, these long delays can lead to actual dead locks between two roles with one team not

being able to continue working, for example without being able to examine the dimensions of new graphics. Sharing partial results early on would improve on that.
Agile teams of program designers generally strive for a short roundtrip time between anyone's change to the system and a visible change in behavior of the system. For example, Extreme Programming proposes to only have one branch of source code and only work on separate branches for a few hours at maximum. Thereby, all changes to the system always become visible to other team members at least at the end of the day. Although, XP promotes a whole team this practice only refers explicitly to source code. Other artifacts relevant to the system are not mentioned. In the end, only program designers can effectively modify the system.
Ideally, every team member would contribute to the system directly. This would still allow for experts to work on their tasks with a special tool set, for example a wireframe editor. However, the resulting artifact should directly have an impact on the system, for example the wireframe could directly determine the layout of a user interface without a program designer translating from the wireframe to source code. Thus, the team would work in a *workshop*-like environment in which the final product is at the center and while every team member would work on it using their specialized tools, they would still all contribute to one result. Further, as they would all work in the same room, they can always see the changes of others and the overall state of the product.
An example of how design teams can implement such a workshop-like environment can be found at Boeing. The team constructing the Boing 777 airplane used a shared 3D model which was always kept up to date with the newest modifications from each team. Further, every team could access it, see the overall state of the airplane design, and examine any interactions between their modifications and modifications of others [McChrystal 2015].
For software development, there is no need for an additional model as the system to be designed is already a digital artifact and could theoretically directly be accessed by every team member.

3. Learning from Program Designers

Exploratory *programming* environments are based on "the conscious intertwining of system design and implementation." [Sheil 1986] They rely on a variety of properties to support divergent and convergent approaches throughout the design process [Trenouth 1991]. However, so far they are based on a very narrow definition of software design as programming. The properties of these environments might actually be generalized to form the conceptual foundation to describe exploratory *software design* environments in which all roles can benefit from these properties. We will first describe the original idea of workflows in exploratory programming environments, the corresponding properties, and illustrate them with two exploratory programming systems: Squeak/Smalltalk [Ingalls 1997] and Lively Kernel [Ingalls 2008, Lincke 2012]. We then describe a generalization of this workflow for "exploratory software design environments" and the adapted properties for such systems.

3.1 Exploratory Programming Environments for Program Designers

The idea of exploratory software development originates from the observations that static, linear development processes do not cope well with complex and often-changing requirements. While the process model was not very explicit, the idea was helpful in shaping programming environments which support the iterative and divergent style of programming, which are called exploratory programming environments [Sheil 1986, Trenouth 1991, Sandberg 1988]. According to a survey by Trenouth, four properties define such systems [Trenouth 1991]:

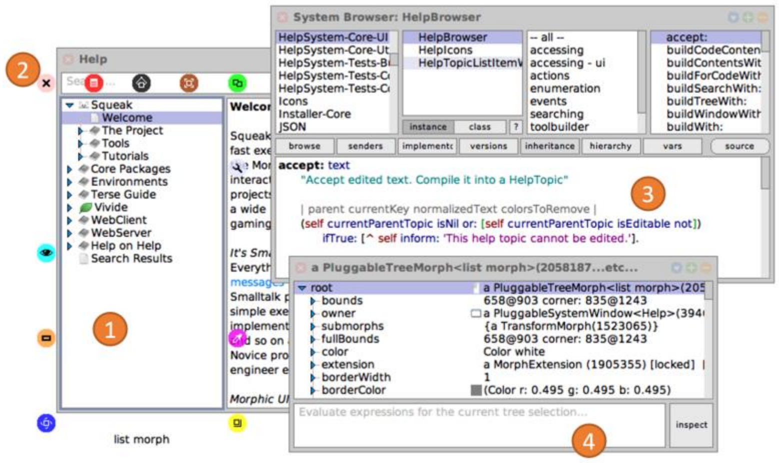
- *Continuously executable*: The product of the exploration process might not only be the software system but also a greater insight which will inform the future process. Thus, a mere static representation of the software as source code is not desirable. The system to be created should ideally be continuously running and usable.
- *Easily extensible*: Programmers should be able to modify the software easily "without adversely affecting existing behavior".
- *Conveniently explorable*: In order to allow the exploration of design alternative, the environment should support the management of alternatives. Consequently, it should allow programmers to quickly switch between the alternatives.
- *Usefully explainable*: The exploratory programming process aims to allow programmers to understand the problem and design space. As such, the environment should provide means to enable programmers to understand the system, for example through state inspection or visualizations of the dynamic system behavior.

3.2 Case "Desktop Development": Squeak/Smalltalk

Squeak/Smalltalk is an exploratory programming environment [Ingalls 1997, Sandberg 1988]. It was designed as a media-authoring and simulation environment. Several versions and extensions explicitly targeted children exploring ideas and models through the environment.

A fundamental principle of the environment is object-orientation which states that every "thing" in the environment is active in the sense that it has some behavior. This behavior is invoked by sending a message to such an object. For example, sending the message capitalized to the object representing the text "smalltalk" would result in the text calculating a capitalized version of itself which is "Smalltalk". The object-orientation is fundamental for Squeak/Smalltalk as *everything is an object* that means all artifacts making up the system such as source code, pictures, sounds, and layout specifications are objects.

Squeak/Smalltalk provides special support for the exploratory creation of graphical objects. All visual elements on the screen are so called *morphs*. A morph can be manipulated through *halos* a kind of meta-menu allowing access to graphical operations such as resizing or rotation (see Figure 3). Through the halos users can also copy a morph and thereby create multiple versions of a morph. Beyond these graphical operations, the halos also give users access to some programming facilities such as defining the behavior of the morph when users click on it.



```
accept: text
    "Accept edited text. Compile it into a HelpTopic"

    | parent currentKey normalizedText colorsToRemove |
    (self currentParentTopic isNil or: [self currentParentTopic isEditable not])
        ifTrue: [^ self inform: 'This help topic cannot be edited.'].
```

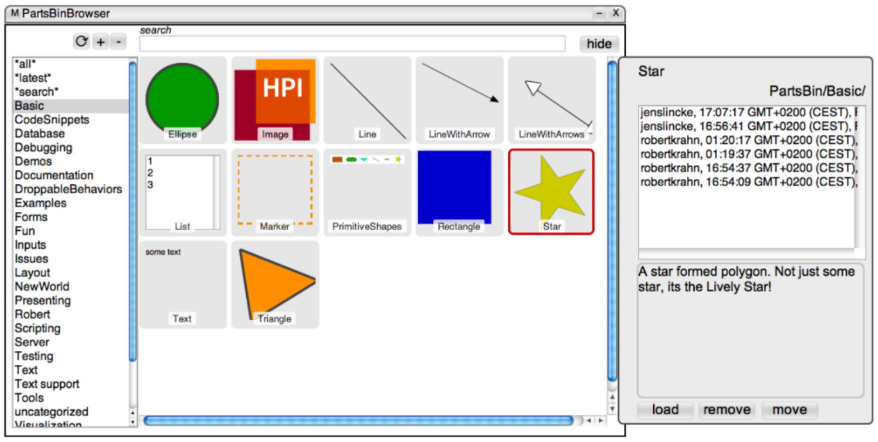
```
root            a PluggableTreeMorph<list morph>(205
  bounds        658@903 corner: 835@1243
  owner         a PluggableSystemWindow<Help>(394
  submorphs     {a TransformMorph(1523065)}
  fullBounds    658@903 corner: 835@1243
  color         Color white
  extension     a MorphExtension (1905355) [locked]
  borderWidth   1
  borderColor   (Color r: 0.495 g: 0.495 b: 0.495)
```

Fig. 3. A screenshot of a list morph on the left (1) with an open halo (2). On the right, the code browser (3) shows parts of the implementation of the help browser tool shown on the left. On the bottom right, an object explorer (4) shows the internal state of the list morph on the left.

Squeak/Smalltalk is also used as a programming system and thus it provides mature tool support for exploratory programming (see Figure 3). Squeak/Smalltalk supports the *continuously executable* features as it allows developers to run applications next to their development tools in the same environment. Programmers can further change the system while it is running without any need to restart it. As Squeak/Smalltalk is a class-based object-oriented environment, it is *easily extensible* trough the addition or modification of classes. The support for *convenient exploration* is available for source code as well as runtime state. Alternative versions of the source code can be managed on a small scale through local versioning of methods. The state of the system can be versioned by saving the current state of the running system into an image file. When the system is loaded from that file it will be in the exact same state it was before. Finally, the environment provides tools to support the *usefully explainable* feature. With the object explorer and inspector tools, programmers can inspect and manipulate any object. The Squeak/Smalltalk debugger enables programmers to stop the execution of any Smalltalk process and inspect and manipulate the state on the stack.

3.3 Case "Web Development": Lively Kernel

Lively Kernel is another exploratory programming environment [Ingalls 2008, Lincke 2012]. As it originates from the Smalltalk tradition of programming systems, it is also object-oriented and exhibits similar tools for exploratory programming. Additionally, it also provides a graphical interface based on morphs and halos.



```
jenslincke, 17:07:17 GMT+0200 (CEST),
jenslincke, 16:56:41 GMT+0200 (CEST),
robertkrahn, 01:20:17 GMT+0200 (CEST),
robertkrahn, 01:19:37 GMT+0200 (CEST),
robertkrahn, 16:54:37 GMT+0200 (CEST),
robertkrahn, 16:54:09 GMT+0200 (CEST),
```

A star formed polygon. Not just some star, its the Lively Star!

Fig. 4. A screenshot of the parts bin in the Lively Kernel environment. Each graphical element can be dragged out and will create a local copy which can be modified by the local user [Lincke 2012].

Lively Kernel, however, allows its users to create the final applications graphical user interface through direct manipulation. After users have assembled their application by combining morphs they can publish their newly assembled graphical object as a *part*. The place where all the published graphical objects are gathered is called the *PartsBin* (see Figure 4) [Lincke 2012]. Every other user of Lively Kernel can instantly see a newly published part and create their own copy by dragging a part out of the PartsBin. Other users can then modify their copies of the part and publish it again as a new part. Thereby, all users can quickly make small changes to parts and share them quickly with other team members.

3.4 Towards Environments for Exploratory Software Design

In the original description of exploratory *programming* environments, the features refer to the relation between programmers, source code, and the running system. However, in a more general sense, they can be applied to any role in the development process to create exploratory *design* environments. Thereby, we move the focus from creating source code to create a running program to creating a variety of artifacts resulting in a software system:

- *Continuously executable*: The software design process should be about creating a working software system which is useful to the user. Thus, everyone participating in the process should be able to execute a current version of the system. This current version should always include their own changes. That means, a content designer creating new sounds should always be able to try their sounds in the environment and interaction designers should always be able to try a new workflow. This property also should hold for the user who should always be able to run a current development version of the system.
- *Easily extensible*: Every team member should be able to easily extend the system in a structured way. A content designer should be able to easily replace or modify content, ideally from within the running system. For example, a graphics designer should be able to modify an icon directly in the running application using graphic editing tools. Consequently, all relevant tools should be included in the environment for every team member. Furthermore, as the design process might unveil new artifacts to be produced the environment should allow for the easy addition of new tools. Such an environment would even allow members of the design team to change the system in the working environment of a user. Thereby, designers can see their effects directly on actual user data and users can immediately see, try, and comment the modifications.
- *Conveniently explorable*: All design activities within the process profit from an interleaving of divergent and convergent approaches. Convergence is naturally part of the process as there is normally only one current version of the software. In contrast, divergence has to be additionally supported. Thus, the environment should allow versioning and branching for all kinds of artifacts produced in the system. Additionally, switching between versions and comparing versions should also be possible for all artifacts. Ideally, the versioning mechanism is the same for all artifacts including source code.
- *Usefully explainable*: The dynamic nature of a running system affects all artifacts produced. For example, an interface might be layouted differently because the displayed name of a user is too long or the display ration of an icon is distorted as the layout specification changes the border of icons on small screens. Thus, for all team members to effectively evaluate their modifications, the environment should provide tools for exploring the dynamic version of the artifact in the running system. For example, these tools should allow

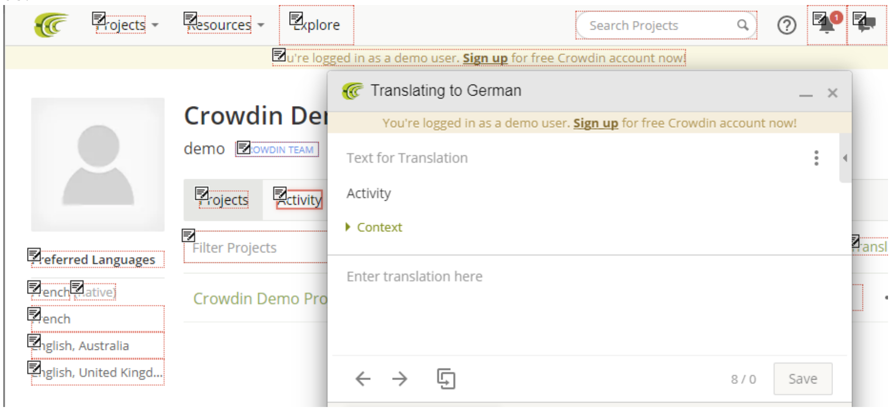
interface designers to determine which user interactions triggered which transitions in the storyboard so that the system ended up in the current state or content designers should be able to see scaling parameters for graphics.

4. Identifying Tools and Environments for Whole Team Software Design

The ideal environment containing all tools that might potentially become relevant for some software project is not possible, due to the variety of domains and constraints for individual projects. However, the idea and the target properties of exploratory software design environments might help in identifying tools and environments which can at least support the collaboration between different roles in a software design team.

4.1 For Individuals: Specific Tools for Specific Tasks

There are a number of tools that integrate the activity of a role with a running instance of the system under design or integrate the artifacts produced by different roles.


u're logged in as a demo user. Sign up for free Crowdin account now!

You're logged in as a demo user. Sign up for free Crowdin account now!

Fig. 5. *A screenshot of the CrowdIn tool for translating text of a webpage within the webpage itself.*

For *content designers*, the tools have to be created as part of the system when the content is specific to the domain or the system. For example, a system for the automatic assessment of insurance claims might have a dedicated editor for business rules. For more general use cases, generic tools are available which integrate the tool and the running system. For example, the CrowdIn tool[4] allows translators to translate a text interactively directly within the webpage where the text is displayed (see Figure 5). Further, to make versioning easy for graphics designers and integrate their artifacts with the source code artifacts of the program designers, tools exist which support the versioning of graphics files. An example is the Kactus tool[5] that integrates the graphics editor SketchApp[6] with the versioning tool Git that is also often used for versioning source code. Thereby, graphic designers and program designers can use the same versioning mechanism and see changes from each other throughout the version history.

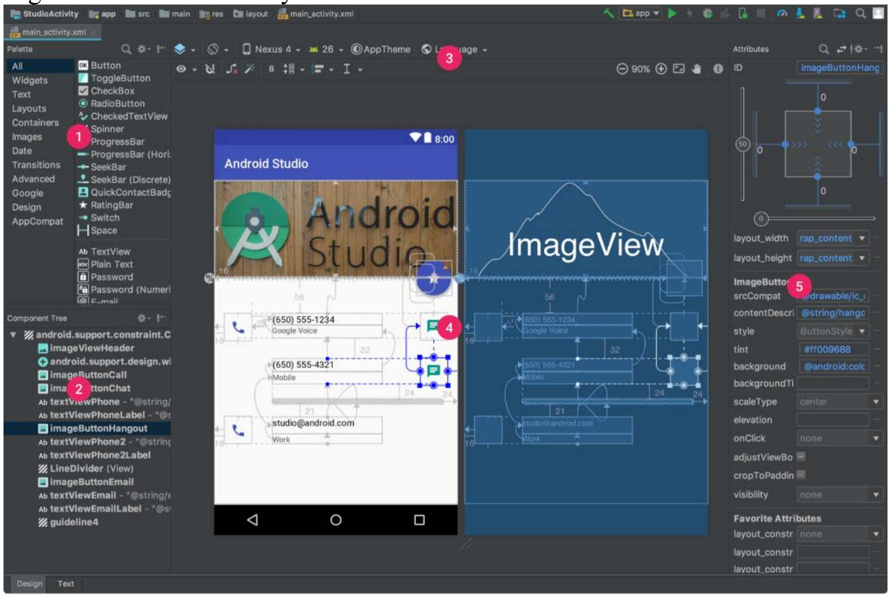


Fig. 6. A screenshot[7] of the Android layout editor showing new widgets (1), the existing layout tree (2), the toolbar (3), the interactive editor (4), and the property view for one element (5)

While *interface design* is often done through mock-ups in graphic editors, it can also be done with tools for creating the actual interface. For example, the Android designer environment includes a layout tool for creating individual screen layouts[8] (see Figure 6). The iOS development environment further supports the creation of

[4] https://web.archive.org/web/20171205114216/https://crowdin.com/page/in-context-localization accessed on 5th of December 2017

[5] https://web.archive.org/web/20171205114339/https://kactus.io/ accessed on 5th of December 2017

[6] https://web.archive.org/web/20171205124457/https://sketchapp.com/ accessed on 5th of December 2017

[7] https://web.archive.org/web/20171205130930/https://developer.android.com/studio/write/layout-editor.html accessed on 5th of December 2017

[8] https://web.archive.org/web/20171205124801/https://developer.android.com/studio/features.html accessed on 5th of December 2017

executable storyboards which define the actual transitions between different views in the resulting mobile application[9]. In both cases, the resulting layout files are directly stored in the directory containing the source code of the application and can be shared with the same tools the source code is shared with.

The degree of participation that is possible for *users* again depends mostly on the kind of system to be designed. For specialized systems the user might actually work next door from the design team and might interact with them in person regularly. For a system with a broader target audience this process has to work differently. However, the integration of giving feedback from within the actual context of usage has been improved by several tools. One example is Instabug[10]. In a mobile application containing Instabug, users can add a new suggestion by shaking the phone. The app will stop, create a screenshot, and ask users for further information on what they would have expected in this situation. A research prototype pushed this mechanism further by converting such suggestions directly into stubs and comments in the source code at the appropriate locations [Kato 2017]. Thereby, users can have a very concrete impact on the artifacts making up the system.

Program design is concerned with the behavior of the system, and thus most tools are close to the system in some aspect. Traditional tools separate the modification of source code from the execution of the system, exploratory tools as described above, integrate the modification of the source code artifacts and the execution of the system (see Figure 3 and 4).

4.2 For Teams: Integrated Tool Environments

For special domains and types of software systems, environments bringing together several roles of the software design team do exist. However, the integration is often based on a thorough understanding of the production processes of the type of software to be created. For example, for a certain type of web applications the requirements and efficient development processes are well known. These environments however, would not work well in situations in which requirements are unknown. Alternatively, design tools might be very well integrated for one particular system which is sometimes done in game development for the design of one particular game.

One example for integrated environments are content management systems (CMS) such as the Drupal system[11]. It provides content designers, users, interface designers, and program designers tools to modify or use the system. For becoming

[9] https://web.archive.org/web/20171205131025/https://developer.apple.com/xcode/interface-builder/ accessed on 5th of December 2017

[10] https://web.archive.org/web/20171205131100/https://instabug.com/ accessed on 5th of December 2017

[11] https://web.archive.org/web/20171205124956/https://www.drupal.org/ accessed on 5th of December 2017

exploratory environments, they are however missing ways to support versioning or branching for comparing different versions. Further, they are specialized on create-read-update-delete (CRUD) systems which are mostly used for managing and publishing digital artifacts.

Another type of systems already integrates many of the relevant tools in one environment: game development environments. Games are complex software systems which require a lot of content design. Consequently, the content and program design are well integrated. An example of such an integration are the development tools for a recent game developed at Nintendo[12]. The integration in their development environment spanned several roles. For example, the interface and program designers were able to see throughout the game world were test users failed most often and could make changes accordingly. For task management, program and interface designers could switch to a task view to see the tasks located next to the relevant location in the world and easily get "get a look at overall completion rates for the game". Further, all content designers were handed the same set of tools: "They created a dedicated software launcher for all the artists to ensure that they were running the same dev[elopment] environment syncing Maya preferences and running automatic tool tests." This focus on integration might be a result of the culture of Nintendo which the game designers described as: "[...] at Nintendo, above all else the most important thing is the fun. This needs to be first and foremost in everyone's mind, regardless of occupation, and they have to tune until the very end to ensure it."[13]

Another environment integrating the activities of several roles is the Home environment which allows the use and modification of productivity tools such as todo lists, e-mail management, or document editing in one environment [Rein 2017]. It is based on Squeak/Smalltalk and Vivide [Taeumel 2014] and thus inherits its exploratory properties. However, the Home environment additionally adds user interface elements which make the system usable as an ordinary desktop system. Users can write emails, create todo items, and store them in a hierarchical ordering system similar to a file system. At the same time all tools can directly be modified using the built-in programming tools without any additional setup or any mode changes. This enables users and program designers to work in the same environment with program designers making live changes in a user's environment or users demonstrating their desired workflows directly within the environment of a program designer.

[12] https://web.archive.org/web/20171205124841/https://medium.com/@gypsyOtoko/the-final-botw-cedec-session-as-far-as-i-know-is-from-the-engineers-botw-project-management-c30f4e42598e accessed on 5th of December 2017

[13] https://web.archive.org/web/20171205124841/https://medium.com/@gypsyOtoko/the-final-botw-cedec-session-as-far-as-i-know-is-from-the-engineers-botw-project-management-c30f4e42598e accessed on 5th of December 2017

5. Conclusion

We described *exploratory software design environments* as a new perspective on the tools used throughout software development teams consisting of program designers, content designers, user experience designers, and users. Taking inspiration from exploratory programming environments, these environments should provide individual team members with more direct feedback from the system to be designed regardless of their role. Consequently, each team member can get an overview of the current state of the system and see the interaction between their modifications and modifications of others. While the creation of one true exploratory software design environment is a wicked problem of itself, individual tools and environments supporting some form of collaboration do exist. By using such tools and environments, teams might be able to grow closer together and create an experience of collaborating while creating a system bringing value to its users.

6. References

- *[Rittel and Webber 1973]* H. Rittel, M. Webber. Dilemmas in a General Theory of Planning. Policy sciences 4(2), 155-169, Springer. 1973
- *[Conklin 2006]* J. Conklin. Dialogue Mapping: Building Shared Understanding of Wicked Problems. Wiley. 2006
- *[DeGrace and Stahl 1990]* P. DeGrace, L. Stahl. Wicked Problems, Righteous Solutions. Yourdon Press. 1990
- *[Beck 2000]* K. Beck. Extreme programming explained: embrace change. Addison-Wesley Professional, 2000.
- *[Sandberg1988]* D. W. Sandberg, Smalltalk and exploratory programming, ACM Sigplan Notices, vol. 23, no. 10, pp. 85-92, 1988.
- *[Arnold 1956]* J.E. Arnold. Problem solving--A creative approach (National Defense University, Publication No. L57-20). Washington, DC: Industrial College of the Armed Forces
- *[Arnold 1959]* J.E. Arnold. Creative Engineering. In W.J. Clancey (Ed.), Creative engineering: Promoting innovation by thinking differently (pp. 59-150). Stanford Digital Repository. http://purl.stanford.edu/jb100vs5754 (Original manuscript 1959)
- *[Buchanan 1992]* R. Buchanan. Wicked Problems in Design Thinking. Design Issues, 8(2), pp. 5-21
- *[McChrystal 2015]* S. McChrystal. Team of Teams. Portfolio / Penguin, 2015.
- *[Ingalls 1997]* D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. ACM SIGPLAN Notices, ACM. 1997
- *[Trenouth 1991]* J. Trenouth. A Survey of Exploratory Software Development. The Computer Journal 34(2), pp. 153-163, Oxford University Press. 1991
- *[Ingalls 2008]* D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, T. Mikkonen. The Lively Kernel: A Self-Supporting System on a Web Page. Proceedings of the Workshop on Self-Sustaining Systems (S3) 2008, Springer. 2008
- *[Lincke 2012]* J. Lincke, R. Krahn, D. Ingalls, M. Röder, R. Hirschfeld. The Lively Parts-Bin -- A Cloud-Based Repository for Collaborative Development of Active Web Content. Proceedings of the Hawaii International Conference on System Sciences (HICSS) 2012. 2012

- *[Rein 2017]* P. Rein, J. Lincke, S. Ramson, T. Mattis, and R. Hirschfeld. Living in Your Programming Environment: Towards an Environment for Exploratory Adaptations of Productivity Tools. In Proceedings of the Programming Experience Workshop (PX/17.2) 2017. ACM. 2017
- *[Taeumel 2014]* M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of Modification and Use in Data-Driven Tool Development. In Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2014. ACM. 2014
- *[Kato 2017]* J. Kato, M. Goto. User-Generated Variables: Streamlined Interaction Design for Feature Requests and Implementations. Proceedings of the Programming Experience Workshop (PX/17) 2017, ACM. 2017