# Feature Crumbs: Adapting Usage Monitoring to Continuous Software Engineering

Jan Ole Johanssen[1], Anja Kleebaum[2], Bernd Bruegge[1], and Barbara Paech[2]

[1] Technical University of Munich, Munich, Germany
{jan.johanssen,bruegge}@in.tum.de
[2] Heidelberg University, Heidelberg, Germany
{anja.kleebaum,paech}@informatik.uni-heidelberg.de

**Abstract.** Continuous software engineering relies on explicit user feedback for the development and improvement of features. The frequent release of feature increments fosters the application of usage monitoring, which promises a broad range of insights. However, it remains a challenge to relate monitored usage data to changes that were introduced by an increment and thereby to a particular specific of a feature.
We introduce *Feature Crumbs*, a lightweight, code-based concept to specify a feature's run-time characteristics. This enables monitored usage data to be allocated to a feature increment. In addition, we analyze the implications for the overall development process. We outline the reference implementation of a platform for collecting, managing, and assessing feature crumbs. We report an evaluation of both the feature crumb concept and the reference implementation in a university capstone course.
Feature crumbs and their changes to the development process contribute to the product quality; they enable feature increment assessment in combination with additional knowledge sources, such as decision knowledge.

**Keywords:** Feature Crumb · Usage Data · User · Usage Monitoring · Process · Agile Development · Continuous Software Engineering.

## 1 Introduction

User feedback is pivotal for software evolution. Continuous software engineering (CSE) [2,7] acknowledges this by relying on early and frequent releases—even with immature prototypes [1]—to retrieve explicit feedback, such as written reviews, from users. This feedback is then turned into change requests for a feature under development. However, this approach is time consuming and dependent on humans [14], leading to a discrete, rather than continuous, source of feedback.

Frequent software releases foster implicit user feedback, namely the application of usage monitoring, enabling a broad research field [17]. Implicit feedback can be collected continuously, without interfering with users, and supports developers in reasoning about how to improve a feature [14]. In contrast to explicit user feedback, which provides the ability for application feature extraction [9], usage data maps to the entire implementation, shown as (B) in Fig. 1. Therefore, one challenge remains in relating usage data to the feature increment specifics.

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

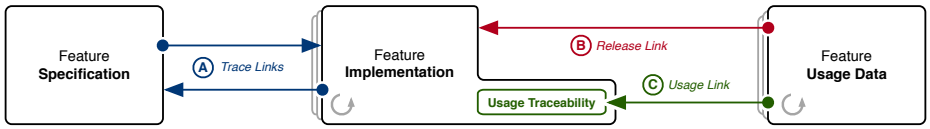


**Fig. 1.** Different link types: feature **specification**s are codified in a feature **implementation** while *trace link*s (A) can be established using feature tags [18]. The implementation is frequently delivered to users who produce **usage data**. Usage data can be mapped to the entire implementation based on a *release link* (B). Represented by feature crumbs, a *usage link* (C) enables actual **usage traceability**, which allows to create a relationship between usage data and implementation specifics.

We introduce the *Feature Crumbs* concept, which allows the allocation of usage data to a feature increment. Similar to requirements traceability, as described by Gotel & Finkelstein [8], feature crumbs create a *usage traceability*, which links usage data to code in a backward direction, shown as (C) in Fig. 1. Feature crumbs represent software sensors that are manually seeded into the application source code to describe the user-centric structure of a feature. Feature crumbs facilitate a detailed evaluation of features; they form the basis for deriving run-time information, such as whether users (a) started, (b) canceled, or (c) finished usage of a feature. Additional information may be recorded at run-time to enrich the assessment of a feature. For example, users' behavioral characteristics aligned with detected feature crumbs might precisely reveal situations in which users are confused. Likewise, decision knowledge regarding the implementation of a feature can be related to the monitored usage data of this feature; such a relationship can be used to assess previous decisions. Ultimately, usage data collected using the feature crumb concept may lead to the discovery of additional requirements or to the refinement of existing requirements.

Along with the introduction of the feature crumb concept, the adaption of usage monitoring to CSE affects multiple parts of the software development process. Thus, we elaborate on new developer capabilities, such as designing run-time feature representations, considerations regarding development artifacts, such as working with branches, and arising needs for additional tool support.

This paper is structured as follows. Section 2 describes requirements for the adaption of usage monitoring to CSE. Section 3 introduces the feature crumb concept and its implications for the software development process. In Section 4, we outline a reference implementation of feature crumbs and report its initial evaluation. We situate our concept in related work in Section 5 and conclude the paper by providing a summary, discussion, and future work in Section 6.

## 2 Requirements

Similar to the integration of decision knowledge into CSE [12], we elicited the following six requirements for the adaption of usage monitoring to CSE. These requirements form the basis for the crumbs concept introduced in Section 3.

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

R1: **Feature Assessment**. Usage monitoring during CSE shall enable developers to assess whether a feature under development was employed by the user. This allows them to only consider usage data produced by a user who actually used a particular feature. Furthermore, performance indicators, such as the cancelation rate of a feature, enable the feature to be assessed.

R2: **Usage Data Allocation**. Feature usage usually involves multiple elements, such as a sequence of buttons that users tap to achieve a goal. A usage monitoring approach shall allow the allocation of observed usage data to a specific step within a feature. This enables precise feature analyses, such as studying time frames between feature steps or relationships to other knowledge sources.

R3: **Feature Flexibility & Comparability**. CSE promotes frequent and rapid releases of new software increments. This requires a usage monitoring concept that allows for a simple extension and flexible replacement of new functional additions to the feature under development. Notably, the concept shall take into consideration that usage data from consecutive feature increments are comparable. This is important to enable the investigation of a feature's evolution.

R4: **Application Range**. The applicability of a concept shall be independent of the product: as CSE is often used in the domain of mobile interactive systems, the concept shall be applicable to graphical user interfaces, but also to computationally intensive code. At the same time, the concept shall be applicable to new user interfaces, such as voice, and consider server-side operations.

R5: **Environment Compatibility**. Development environments are a heterogeneous composition of management approaches, software processes, tools, and platforms. The concept shall be compatible with existing environments and not impose an additional burden upon developers. For example, it should make no difference whether a project organizes features with user stories or scenarios.

R6: **Effort & Learnability**. Usage monitoring shall be easy for developers to learn and apply. In particular, the concept shall impose a minimal cognitive load upon developers responsible for adding and maintaining the means for usage monitoring to guarantee its seamless and continuous application.

## 3 Feature Crumb Concept

Following the above-mentioned requirements, we present an object model of the feature crumb concept in Section 3.1 and describe its implications for the development process in Section 3.2. Thereby, we highlight its lightweight character.

### 3.1 Object Model

The major entities and their relations are depicted in Fig. 2. A `Feature` has a `name` and a `description`. A `Path` is uniquely defined for every feature. It consists of a sequence of `steps`, which represents a sorted array of `Crumb`s to specify the flow of crumbs needed to complete a feature. As a feature is extended or updated, crumbs can be `add`ed to or `remove`d from a path. If the introduced changes are major, both activities increment the path `version`.

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

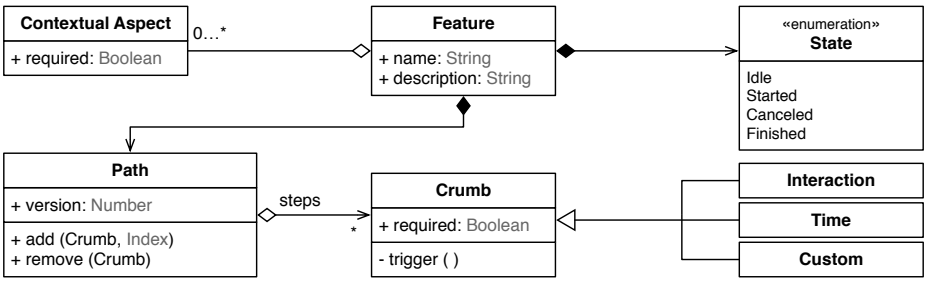


**Fig. 2.** Object model representing the feature crumb concept as a UML class diagram.

If a feature is composed of optional steps, the respective crumb's attribute `required` might be set to *false*. We distinguish between different classes of crumbs, which all share the capability to `trigger` an event, each under a certain condition: an `Interaction` crumb might be triggered by any interaction or event that can occur during the run-time of an application. A `Time` crumb defers its trigger call until a specified time frame is passed. `Custom` crumbs await any individually-designed conditions, such as the input of a pre-defined string into a text field, before triggering an event. A feature's `State` relates to its observed execution. Based on the recorded events triggered by crumbs and given the feature path, we distinguish the following feature states: (a) *Idle*, if the feature has never been initiated by the user; (b) *Started*, if the first crumb of the feature path sequence was detected; (c) *Canceled*, if the assessment of a path is stopped due to a certain criterion; or (d) *Finished*, if all crumbs that are part of the sequence were detected sequentially. Eventually, `Contextual Aspects` describe conditions that need to be met to start the feature path observation. These may be pre- or post-conditions, such as an external event or the availability of certain user data. Contextual aspects can be optional. An informal representation of the feature crumb concept is sketched in Fig. 3 to illustrate objects shown in Fig. 2.

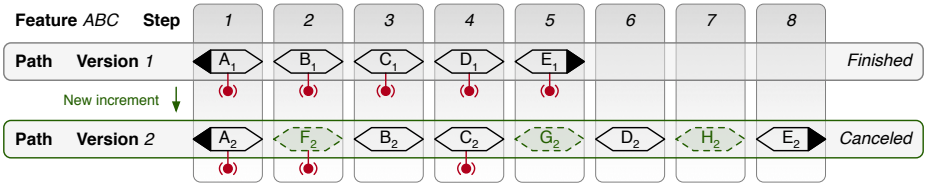


**Fig. 3.** A `Feature` entitled *ABC* that was improved once. Thus, `Path` was updated with a new `version` number. Version *1* consists of five feature `Crumbs`, which are depicted as a hexagon while black corners signal either the first or last crumb of a `steps` sequence. Version *2* introduces three new crumbs (dashed, green border) that occupy steps that were previously allocated to other crumbs. The red antenna signals the call of the `trigger` method; while this means for version 1 that the feature has been executed properly (*Finished*), version 2 was *Canceled*, since $C_2$ was triggered before $B_2$.

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

| Phases | Requirements Elicitation | Implementation | Testing | Deploying | Usage Monitoring |
|---|---|---|---|---|---|
| **Capabilities** | Elicit and implement requirements, design run-time feature representation * | | Design and assess test cases, analyze crash reports | | Evaluate monitored data * |
| **Artifacts** | User Story, Scenario * | Branches and Commits * | Test Cases | Crash Reports | Feature Crumbs * |
| **Tools** | Issue & Knowledge Management System | Version Control System | Integration System | Delivery and Distribution System | User Understanding System * |

**Fig. 4.** Outline of major capabilities, artifacts, and tools across the phases of a software development process; partially drawn and combined from [2] and [3]. Individual entries are interweaved and may be based on their predecessor. This is relevant for the entries related to usage monitoring, which are written in red font and followed by a star (*).

### 3.2 Implications for the Development Process

The introduction of feature crumbs and the adaption of usage monitoring to CSE impacts the overall development process. This is because usage monitoring is situated at the end of a development cycle as it verifies the product. In Fig. 4, we outline the major capabilities required by developers, resulting artifacts, and supportive tools when cycling through a software development process.

A software development process depends upon the elicitation and management of requirements for a feature. A feature might be described as a use case or as a scenario, while the latter represents a concrete instance of this use case [3]. Given their lightweight character, feature crumbs fit into different concepts of how a feature is designed, managed, or tracked during design-time. Notably, given the similar structure of subsequent events, feature crumbs promote scenarios, highlighting their usefulness for collecting and analyzing usage data [15].

The selection of a branching strategy affects the work with feature crumbs: we propose relying on a branching model that uses feature branches for implementation work [13] to allow the integration of knowledge into CSE [11]. Feature crumbs can be added only to feature branches to maintain feature atomicity and to avoid interference with the development of other features.

During requirements elicitation and feature implementation, developers need to be able to design a run-time representation of the feature on which they are working to make it evaluable during usage monitoring. This requires additional effort during requirements elicitation and implementation: adding feature crumbs is similar to writing test cases—for which developers have to reflect on the feature before coding [5]—to be able to verify the implementation.

Only after a software increment is delivered into a target environment, usage monitoring based on feature crumbs can be performed. To manage and analyze usage data, tool support in the form of a user understanding system is required. We provide one aspect of such as system—with a focus on the collection, management, and assessment of feature crumbs—in the following Section 4. This enables developers to evaluate monitored data and reflect upon the product quality.

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

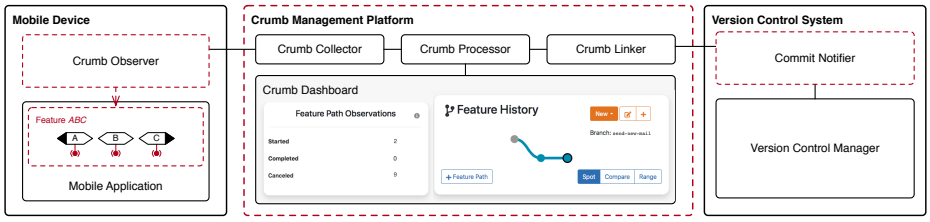


**Fig. 5.** The reference implementation including the major systems. Dashed, red borders indicate components created or added to enable the feature crumb concept.

## 4 Reference Implementation and Initial Evaluation

We developed a **reference implementation** for the feature crumb concept to demonstrate its feasibility. In Fig. 5, we outline the three major systems: First, a framework for *Mobile Devices* allows developers to integrate crumbs for a feature. This framework also includes a *Crumb Observer* that reports triggered crumbs during application run-time. Second, we developed a *Crumb Management Platform*, which is the main interaction point for developers to work with the crumbs. It receives crumbs from applications via a *Crumb Collector*. The *Crumb Processor* is in charge of connecting received crumbs and the feature path information. The result, i.e., the feature state, is visualized on a *Crumb Dashboard* which further enables developers to define a feature path for a specific software increment that is released to users. The *Crumb Linker* is in charge of creating a relationship between a feature and a release that is uniquely identified by a code commit. The commit information is provided by a *Commit Notifier*, which represents the third aspect of the implementation. Based on a webhook system, this component informs the crumb management platform about new commits.

We ran an **initial evaluation** of the feature crumb concept and the reference implementation according to two variables of the technology acceptance model: the *perceived usefulness* (U) and the *perceived ease-of-use* (E) [6]. As a sample of prospective users, we relied upon nine students, each performing a usability representative role in a project within a university capstone course in which up to 100 students work on real industry projects [4] over the course of two months. Regarding *U*, all projects were able to define one or more features, including a feature path and crumbs. Users reported useful aspects, such as being able to *determine the repetition of a feature* to detect important steps, or to *assess if a customer was able to complete a feature* to detect if a navigational path might be unclear. At the same time, one user reported that their *app did not include much user interaction*, which limits the usefulness of feature crumbs; still, they would have been able to measure implementation internals. Regarding *E*, we obtained diverse feedback: the initial manual addition of commits was perceived as inconvenient, which led to the development of the commit notifier. Some users reported that the registration of a feature path was cumbersome and error-prone. One user emphasized *increased effort to integrate crumbs into code.*

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

## 5 Related Work

The feature crumb concept forms a lightweight task model to describe user interactions and shares similarities with the *ConcurTaskTrees* specification [16], which introduces four task types: a *user*, an *abstract*, an *interaction*, and an *application* task. Given their manifestation in code, feature crumbs do not distinguish *who*, e.g., the system or user, triggered them. Therefore, in Fig. 3, we apply the same symbol—the hexagon, which is used for application tasks in [16]—for all classes of feature crumbs (Fig. 2). Generally, feature crumbs focus on applications developed during fast-cycled processes, such as CSE. Moreover, they depict a linear path, rather than hierarchies or logical relationships.

To implement actual run-time observation, development environments[3] rely on code additions, similar to feature crumbs, that enable developers to create success and cancel paths. These concepts are, however, for debugging and profiling applications. Platforms[4] that apply such concepts for usage monitoring only address single events and do not promote the addition of other knowledge types.

There exist various approaches that utilize usage data for software evolution. For instance, UI-Tracer [10] supports developers in comprehending a software system by automatically identifying source code that is related to user interface elements. Similarly, feature crumbs can be understood as traces that relate user elements with their implementation; however, our concept is not limited to user interface elements. Furthermore, feature crumbs describe features to collect usage data, which can be linked to other knowledge types, such as decision knowledge.

## 6 Conclusion

In this paper, we have presented feature crumbs, a lightweight, code-based concept to describe a feature with the goal of adapting usage monitoring to CSE. We have summarized six requirements and implications for the development process. We have outlined a reference implementation to demonstrate the feasibility of collecting, managing, and assessing feature crumbs. We have reported an initial evaluation of both the concept and the implementation in a university context.

**Discussion.** The implementation and evaluation have indicated that feature crumbs promote usage monitoring in CSE. Now, we face two major areas for discussion. First, the focus of a feature: while we provide the means for feature definition and tracking, we observed that it is difficult to decide where to seed crumbs, i.e., the first crumb. We consider providing a guideline to developers. Second, the platform usability: developers need different functionalities. Discussion is required to select important ones and make them easily accessible.

**Future Work.** We plan to improve the feature crumb management platform towards a comprehensive user understanding system and to continue evaluating it to investigate developers' acceptance. A long-term goal is to add multiple knowledge sources that can be better assessed when relying on feature crumbs.

[3] https://developer.apple.com/videos/play/wwdc2018-405/?time=1097
[4] https://docs.microsoft.com/en-us/appcenter/analytics/event-metrics

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.

**Acknowledgments.** This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution. We thank Jan Philip Bernius and Lara Marie Reimer for their excellent work on the development of the reference implementation and the participants of the university capstone course for their feedback.

## References

1. Alperowitz, L., Weintraud, A.M., Kofler, S.C., Bruegge, B.: Continuous prototyping. In: 3rd Int. Workshop on Rapid Continuous Softw. Eng. pp. 36–42 (2017)
2. Bosch, J.: Continuous Software Engineering: An Introduction. Springer (2014)
3. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall Press, 3rd edn. (2010)
4. Bruegge, B., Krusche, S., Alperowitz, L.: Software engineering project courses with industrial clients. ACM Transactions on Com. Edu. **15**(4), 17:1–17:31 (2015)
5. Crispin, L.: Driving software quality: How test-driven development impacts software quality. IEEE Software **23**(6), 70–71 (2006)
6. Davis, F.D., Bagozzi, R.P., Warshaw, P.R.: User acceptance of computer technology: A comparison of two theoretical models. Mangagement Science **35**(8), 982 – 1002 (1989)
7. Fitzgerald, B., Stol, K.J.: Continuous software engineering: A roadmap and agenda. Journal of Systems and Software **123**, 176–189 (2017)
8. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proc. of IEEE Int. Conf. on Requir. Engineering. pp. 94–101 (1994)
9. Guzman, E., Maalej, W.: How do users like this feature? a fine grained sentiment analysis of app reviews. In: IEEE 22nd Int. Requir. Eng. Conf. pp. 153–162 (2014)
10. Hebig, R.: UI-tracer: A lightweight approach to help developers tracing user interface elements to source code. In: Software Engineering und Software Management 2018. pp. 225–236 (2018)
11. Johanssen, J.O., Kleebaum, A., Bruegge, B., Paech, B.: Towards a systematic approach to integrate usage and decision knowledge in continuous software engineering. In: Proc. of the 2nd Workshop on Continuous Softw. Eng. pp. 7–11 (2017)
12. Kleebaum, A., Johanssen, J.O., Paech, B., Bruegge, B.: Tool support for decision and usage knowledge in continuous software engineering. In: Proceedings of the 3rd Workshop on Continuous Software Engineering. pp. 74–77 (2018)
13. Krusche, S., Alperowitz, L., Bruegge, B., Wagner, M.O.: Rugby: An agile process model based on continuous delivery. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. pp. 42–50 (2014)
14. Maalej, W., Happel, H.J., Rashid, A.: When users become collaborators: Towards continuous and context-aware user input. In: Proc. of the ACM SIGPLAN Conf. Companion on Object Oriented Program. Syst. Lang. and Appl. pp. 981–990 (2009)
15. Nielsen, J.: Scenarios in discount usability engineering. In: Carroll, J.M. (ed.) Scenario-based Design, pp. 59–83. John Wiley & Sons, Inc. (1995)
16. Paterno, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models, pp. 362–369. Springer US (1997)
17. Ros, R., Runeson, P.: Continuous experimentation and a/b testing: A mapping study. In: 4th Int. Workshop on Rapid Continuous Softw. Eng. pp. 35–41 (2018)
18. Seiler, M., Paech, B.: Using tags to support feature management across issue tracking systems and version control systems. In: 23rd Int. Working Conf. on Requir. Eng.: Foundation for Softw. Quality. vol. LNCS 10153, pp. 174–180. Springer (2017)

This is the author-created version of the work published at the *19th International Conference on Product-Focused Software Process Improvement*. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-03673-7_19.