DebtFlag: Technical Debt Management with a Development Environment Integrated Tool

Johannes Holvitie, Ville Leppänen
TUCS - Turku Centre for Computer Science
&
Department of Information Technology
University of Turku
Turku, Finland
{jjholv,ville.leppanen}@utu.fi

***Abstract*—In this paper, we introduce the *DebtFlag* tool for capturing, tracking and resolving technical debt in software projects. *DebtFlag* integrates into the development environment and provides developers with lightweight documentation tools to capture technical debt and link them to corresponding parts in the implementation. During continued development these links are used to create propagation paths for the documented debt. This allows for an up-to-date and accurate presentation of technical debt to be upheld, which enables developer conducted implementation-level micromanagement as well as higher level technical debt management.**

***Index Terms*—Technical debt, technical debt management, source code assessment, source code analysis, DebtFlag.**

I. Introduction

Ward Cunningham was the first to coin the term technical debt [1]. In his technical report deviation from the design incurs the principal of technical debt. Refactorization pays it back. Development on top of the principal counts as technical debt's interest and hindered development constitutes as paying interest. Like with its financial counterpart: technical debt is acceptable as long as its payback is managed.

Technical debt management is a rather new research area, interested in introducing control over technical debt by providing projects with means to identify, assess and payback technical debt. Seaman et al. [2] raise the availability and clarity of technical debt information as one of the key factors to successful technical debt management.

Highly complex, self-emergent and frequently changing software products are a challenging ground for technical debt information production. Automatic approaches are capable of accommodating the change rates that major development projects introduce, but their reliance onto statically predefinable models make them incapable of modeling the entire requirement space [3], [4]. Manual approaches capture the entire space [5], [6] but due to their nature they consume a large amount of development resources which disallows their frequent use.

The mechanism we introduce has been designed to exploit the benefits of both aforementioned assessment approaches. *DebtFlag* links structured observations about technical debt to related parts in the software implementation. The structure does not limit the declaration but it retains an equivalency between the entries that makes automatic updates possible.

When used in software projects, *DebtFlag* captures technical debt through lightweight documentation tools that integrate into the development environment. It tracks the propagation of technical debt by building dependency trees for associated software implementation parts. *DebtFlag* allows to resolve technical debt by supporting its management on two different levels. Developer conducted micromanagement through maintaining an implementation level representation of technical debt and project level management by making *DebtFlag* cater for the information needs of higher level approaches.

II. *DebtFlag* Mechanism

The *DebtFlag* mechanism has been designed to be compatible with different software implementation techniques. This section provides a description for it, explaining the documentation structure for technical debt, the requirements and functionality for automation and describes how these pursue two different technical debt management approaches.

A. Structure of Documented Technical Debt

The structure for documenting technical debt with the *DebtFlag* mechanism is based onto the documentation structure introduced as part of the Technical Debt Management Framework (TDMF) [7], [8] by Seaman et al.. This structure is extended in the *DebtFlag* mechanism in order to decompose entries into reusable components as well as to properly present technical debt at the implementation level.

The Technical Debt Management Framework is a three parted approach on managing technical debt in software projects. It relies onto a Technical Debt List (TDL) constructed in the first, technical debt identification, part. The list is populated with Technical Debt Items (TDI), which correspond to single atomic occurrences of technical debt in the project.

A Technical Debt Item documents and upholds a set of information [7]. A description explains the debt's type, location and reasoning for its acquirance. An estimate for the debt's principal indicates how much resources are required to pay it back – to make this partition fully adhere to the design. While, an interest estimates the probability and amount of extra work this principal can cause to future development.

```
Technical Debt List
> Technical Debt Item #1
    > DebtFlag #1
        - Time and Date
        - Author
        - Location
        - Description
        - Type Declaration
            > Technical Debt Type #1
                - Name
                - Description
                - Context
                - Propagation rules
            > Technical Debt Type 2
                ...
    > DebtFlag #2
        ...
    ...
> Technical Debt Item #2
    ...
...
```

Fig. 1. The *DebtFlag* mechanism's documentation structure for technical debt

This structure is extended with an additional level. In the *DebtFlag* mechanism a TDI can consist out of a single or multiple *DebtFlag* elements. A *DebtFlag* element is a link between a technical debt observation and an implementation part defined by the technique. For example a package, a class or a method in object-oriented technologies.

By allowing TDIs to be constructed with *DebtFlag* elements we want to preserve a degree of freedom: the description of a TDI's location is not limited to a single area predefined by the implementation technique, but rather it can encompass an unlimited amount and combination of them. This makes it possible for a single TDI to have unrestricted propagation capabilities. At the same time, the possibility to use *DebtFlag* elements as equals to TDIs is kept.

The attributes of a *DebtFlag* element are inherited from a TDI and consist out of a time and a date, an author, a type, a location and a description. The time and date indicate when the observation was made. The author corresponds to the responsible developer. The location to an element in the implementation, as described by the previous paragraph. The type attribute has been extended and made component based. The *DebtFlag* element type consists out of a set of technical debt types. These types can be either predefined or created during use. A technical debt type documents a name, a description, a context and a propagation rule set for it. The type name and the description are self-explanatory. The context attribute binds this type to a certain implementation context - for example a programming language. The propagation rule set is used for declaring the propagation capabilities for this type of technical debt, its principal and the effect the propagation has onto the accumulation of interest.

B. Automation of Technical Debt Propagation

Dynamic functionality of the *DebtFlag* mechanism relies onto being able to automate two processes. The process of identifying source points for the propagation of technical debt and the process of propagating the technical debt according to rule sets and dependencies in the implementation.

Capturing technical debt with the *DebtFlag* mechanism corresponds to creating TDIs by forming collections of *DebtFlag* elements. As stated, a *DebtFlag* element represents a link between a technical debt observation and a corresponding implementation part. Depending onto the used implementation technique, an implementation part can contain several points capable of forming dependencies. Dependencies carry technical debt and increase its interest [9]. Projection of technical debt onto the implementation is thus dependent onto being able to identify those source components responsible for propagating technical debt. This functionality is dependent onto information about the used implementation technique.

After acquiring the source implementation components for technical debt, the *DebtFlag* mechanism completes the projection by propagating technical debt through dependencies while following a possible rule set. The process takes a source component and goes through the *DebtFlag* elements associated with it. For each *DebtFlag* element the technical debt type declared for it determines the rules for its propagation. According to these the *DebtFlag* mechanism associates the *DebtFlag* elements with implementation components that are directly or indirectly dependent onto their source components.

Figure 2 presents a simplified example scenario and how the aforementioned two processes function. Classes `A` and `B` both contain two methods `A.a` and `A.b` as well as `B.c` and `B.d` respectively. Method `B.d` is dependent onto method `B.c` which again is dependent onto method `A.a`. Two *DebtFlag* elements are created. First one for the entire class `A` and a second one for just the method `B.c`. From the aforementioned

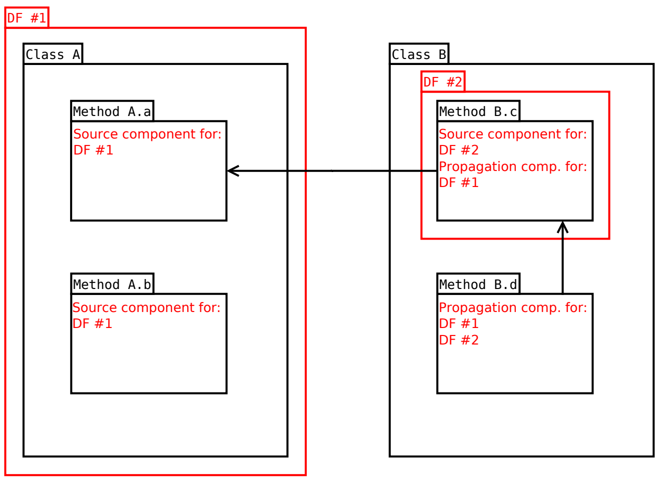


Fig. 2. Propagation of technical debt in the *DebtFlag* mechanism

processes, the first one now returns methods `A.a` and `A.b` as the source points for the first *DebtFlag* element and method `B.c` for the second element. The second process can now be called with these three source points. For method `A.a` it will return a dependency tree containing methods `B.c` and `B.d`, for method `A.b` an empty tree and for method `B.c` a dependency tree containing method `B.d`. Based onto this information and the propagation rules associated with each *DebtFlag* element, the *DebtFlag* mechanism associates each component in Figure 2 with captured technical debt.

C. Technical Debt Management

The *DebtFlag* mechanism is designed to support technical debt management in two different forms. Project and organization level management through supporting the TDMF and implementation level micromanagement by decomposing and projecting technical debt onto the implementation.

The Technical Debt Management Framework allows a variety of processes to be used for managing technical debt. The functionality of the TDMF is dependent onto the existence of the Technical Debt List. The *DebtFlag* mechanism has been designed so as to be able to efficiently construct and maintain the TDL.

The TDL is populated with Technical Debt Items that are formed from *DebtFlag* elements. The *DebtFlag* elements inherit their basic attributes from the TDI and thus answer to them as a set. In order to maintain the TDL during continued development the *DebtFlag* elements are designed not to prompt estimates about technical debt's principal and interest. Rather, these estimates are based on the current number of *DebtFlag* elements forming a TDI, the technical debt types they are associated with and the information about their propagation. Essentially, the used propagation rule set defines the technical interest for all technical debt items. We discuss the importance of this in Section IV-C and how we intend to take this into account in Section V-A.

Micromanagement of technical debt is supported by creating and maintaining a presentation for it on the implementation level. As technical debt is decomposed into source propagation points and propagated onwards according to monitored dependencies the produced information enables the debt to have a presentation at these points. By using for example visualization and restriction (see Section III) the developer can be made aware of otherwise unnoticeable technical debt and its properties. We discuss our expectations for technical debt aware software development in Section IV.

III. *DebtFlag* TOOL

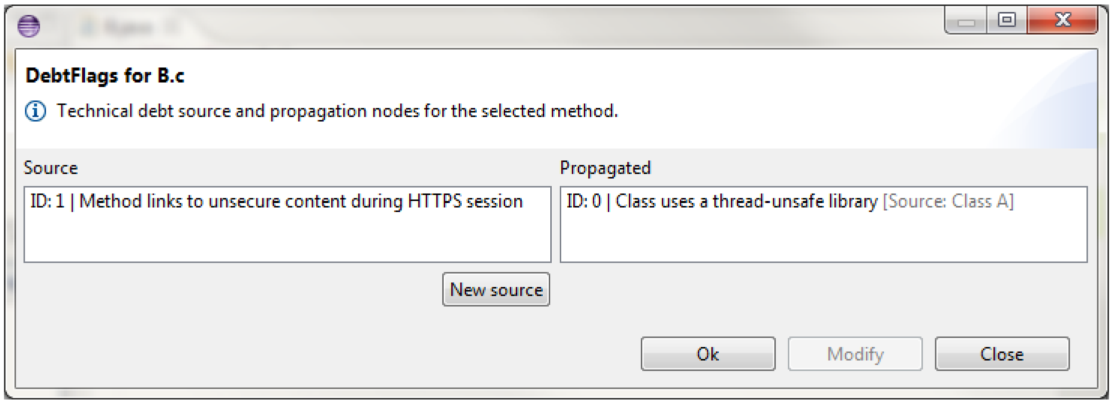
The first implementation of the *DebtFlag* mechanism was designed to support the Java programming language. The *DebtFlag* tool is a two parted system consisting out of a plug-in for the Eclipse Integrated Development Environment (IDE) [10] and a separate web application. The *DebtFlag* plug-in is responsible for capturing technical debt through the development environment, tracking its propagation and supporting the micromanagement approach. The web application provides a dynamic presentation of the Technical Debt List compiled from information produced with the *DebtFlag* plug-ins.

A. *DebtFlag Plug-In*

Eclipse is a popular IDE used especially for developing in the Java language. It is built on the concept of plug-ins and by implementing the first part of the *DebtFlag* tool as such, we are capable of integrating the documentation tools into the development environment, identifying the source implementation elements for technical debt, propagating the debt according to dependencies and building a representation of technical debt on the implementation level.

1) Capturing Technical Debt: Capturing technical debt using the *DebtFlag* plug-in is started by interacting with a Java element. Eclipse provides multiple views to the Java element hierarchies it is used to modify. Valid Java elements from the *DebtFlag* plug-ins perspective range from a package to a class member, such as a method or a global variable. All of these can serve as the location for a *DebtFlag* element and trigger the documentation process.

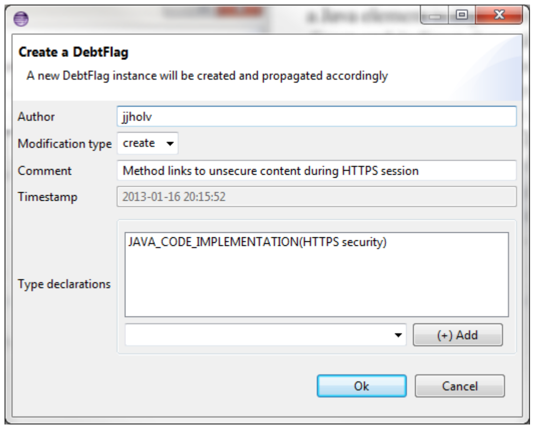
Figure 3 depicts the listing dialog triggered by interaction with a Java element. In this case, the element is method `c` from class `B` and the dialog displays two *DebtFlag* element listings for it. The first one contains those instances where the


DebtFlags for B.c

Technical debt source and propagation nodes for the selected method.

Source	Propagated
ID: 1 | Method links to unsecure content during HTTPS session	ID: 0 | Class uses a thread-unsafe library [Source: Class A]

Fig. 3. The *DebtFlag* element list triggered through interaction with a Java element


Create a DebtFlag
A new DebtFlag instance will be created and propagated accordingly

Author	jjholv
Modification type	create
Comment	Method links to unsecure content during HTTPS session
Timestamp	2013-01-16 20:15:52
Type declarations	JAVA_CODE_IMPLEMENTATION(HTTPS security)

Fig. 4. The creation of a new *DebtFlag* element event

debt's source location is the current Java element, while the second one contains those propagated to it via dependencies. From here the user can choose to either create a new *DebtFlag* element - having this Java element as its source - or to modify any of the displayed *DebtFlag* elements.

Events forming the life span of a *DebtFlag* element are predetermined. It starts with a *create* event, followed by any number of *modify* events and ends with a *resolve* event. Figure 4 depicts the dialog for producing *DebtFlag* element events. In this case, the "create" event of a new *DebtFlag* element for the aforementioned method `B.c`.

The *DebtFlag* element event dialog (see Figure 4) captures the author, the modification event type, the comment, the time and the type declaration for a *DebtFlag* element instance. The location information is not prompted as this corresponds to the Java element triggering this dialog. The author, the modification event type and time attribute are prefilled according to system wide information. The type declaration part shows the currently selected technical debt types used for defining this *DebtFlag*, the accumulated library of available types and the possibility to add new ones. The comment attribute is a free text form intended for providing the reasoning for the event.

As mentioned in Section II-A, a *DebtFlag* element's type is a collection of technical debt types. When defining a new *DebtFlag* element event, if the provided library does not contain a suitable type combination, new ones can be created. Figure 5 depicts the dialog for creating a new technical debt type. It captures the type's name, description, propagation capabilities, context and threshold. Currently, the propagation capability of a type is a binary option for either declaring that this type can propagate through dependencies – increasing the debt's interest – or that it is confined inside the Java element. The threshold attribute is used to communicate the severity of the type by declaring how many dependencies can be formed to elements carrying it before additional measures are enforced (see Section III-A2). A lower threshold can indicate a high principal, rapidly growing interest, probable realization or a combination of these.

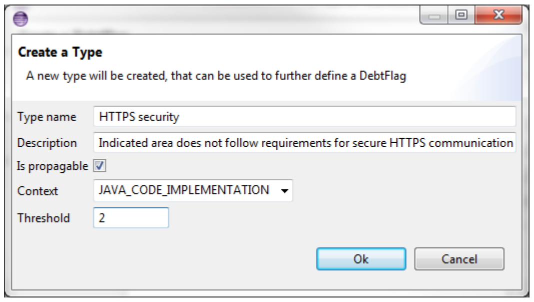
2) Implementation Level Representation of Technical Debt: The *DebtFlag* plug-in builds an implementation level representation of technical debt in order to support its micromanagement. The representation uses visualization and restriction in the Eclipse IDE for indicating the presence of technical debt. The information, the representation is based on, is produced with the decomposition and propagation processes described in Section II-B and implemented using the Eclipse Java Development Tools (JDT). The JDT is a core Eclipse plug-in which generates information about Java implementation structures.

The *DebtFlag* plug-in modifies the visual appearance of each Java element that has either source or propagated debt. The effect technical debt has on the visual representation of a Java element is dependent onto two matters. The number of direct and indirect dependencies coming to an element and the technical debt types associated to this element. The end results lead to four representation categories for technical debt. Each category excludes the others and has priority over those mentioned before it.

The *source* category indicates that a Java element is the source point for technical debt propagation (default illustration with light red color). The *propagated* category indicates that a Java element is on the propagation path of technical debt – that it is directly or indirectly dependent to a *source* Java element (default illustration with light green color). The *source and propagated* category combines the former groups (default illustration with orange color). Finally, the *debt over threshold* category is used to indicate that the number of dependencies to this Java element exceeds a threshold defined for its technical debt (default illustration with dark red color).

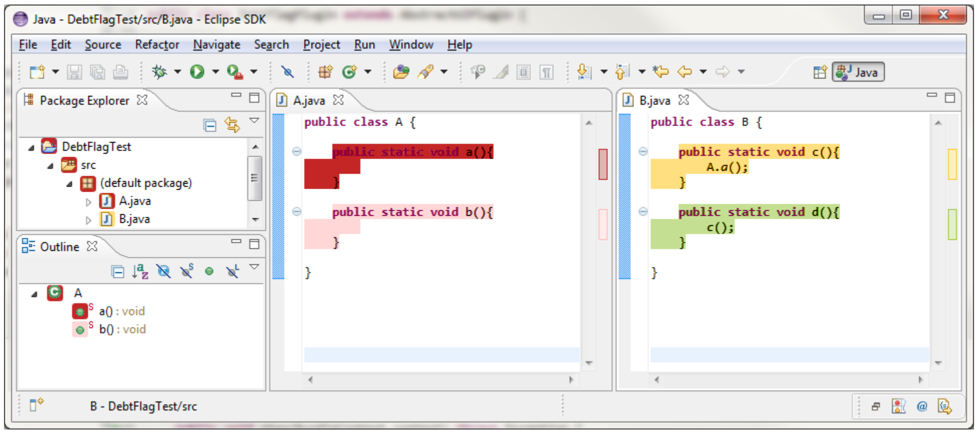
Figure 6 shows the structure from Figure 2 implemented in Java using the Eclipse IDE while employing the *DebtFlag* plug-in to make two technical debt declarations. The first one is made for class `A` with a technical debt type having a threshold value of one (1). This has resolved into two source points for propagation: methods `A.a` and `A.b`. The second declaration has been made directly for method `B.c` with a technical debt type having a threshold value of two (2).

In Figure 6 method `B.d` is dependent onto method `B.c` which again is dependent onto method `A.a`. As this exceeds the threshold value of `A.a` the *DebtFlag* plug-in uses the *debt*


Create a Type
A new type will be created, that can be used to further define a DebtFlag

Type name	HTTPS security
Description	Indicated area does not follow requirements for secure HTTPS communication
Is propagable	✓
Context	JAVA_CODE_IMPLEMENTATION
Threshold	2

Fig. 5. The creation of a new *DebtFlag* element type



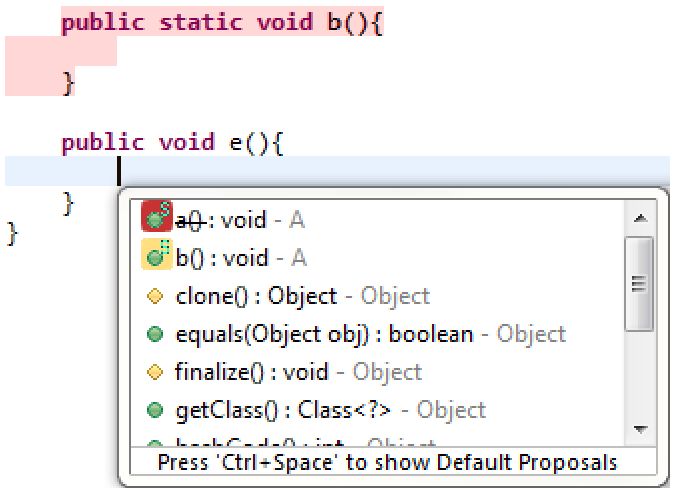
```
public class A {

    public static void a(){
    }

    public static void b(){
    }

}
```

```
public class B {

    public static void c(){
        A.a();
    }

    public static void d(){
        c();
    }

}
```

Fig. 6. The main view of the Eclipse IDE, while using the *DebtFlag* plug-in to manage technical debt instances

over threshold category in its visualization. From the same dependency chain, method `B.c` also had its own technical debt declared for it leading to having the *source and propagated* visualization, while method `B.d` only carries propagated debt and hence has the *propagated* visualization. Finally, method `A.b` as the other source point for propagating technical debt in class `A` has the visual appearance of the *source* category.

The other component in forming the representation is restriction. Figure 7 shows the Eclipse content-assist. It provides developers with dynamic content assistance depending onto the cursor's position in the editor. The figure in question shows the content assistant opened when the cursor is inside class `A` (see Figure 6). The restriction is applied here in the form of a strike through over the method `A.a`. This optional feature of the *DebtFlag* plug-in ensures that no new dependencies are introduced for elements that have their threshold crossed by disallowing their use.



```
    public static void b(){

    }

    public void e(){
        |
    }
}
```

~~a()~~ : void - A
b() : void - A
clone() : Object - Object
equals(Object obj) : boolean - Object
finalize() : void - Object
getClass() : Class<?> - Object
Press 'Ctrl+Space' to show Default Proposals

Fig. 7. The Eclipse content-assist, affected by the *DebtFlag* plug-in

Each *DebtFlag* plug-in works on its own set of *DebtFlag* elements. This set corresponds to elements associated with the latest version checked out to the Eclipse IDE and the additions and modifications made to them afterwards. When changes to the implementation are committed through the version control system, the *DebtFlag* plug-in communicates with the Eclipse Team plug-in. This allows the *DebtFlag* plug-in to gain information about possible conflicts between implementation versions and their resolutions. According to this information an individual *DebtFlag* plug-in builds a *DebtFlag* element set that corresponds to the new version and inserts it to a database.

B. *DebtFlag Web Application*

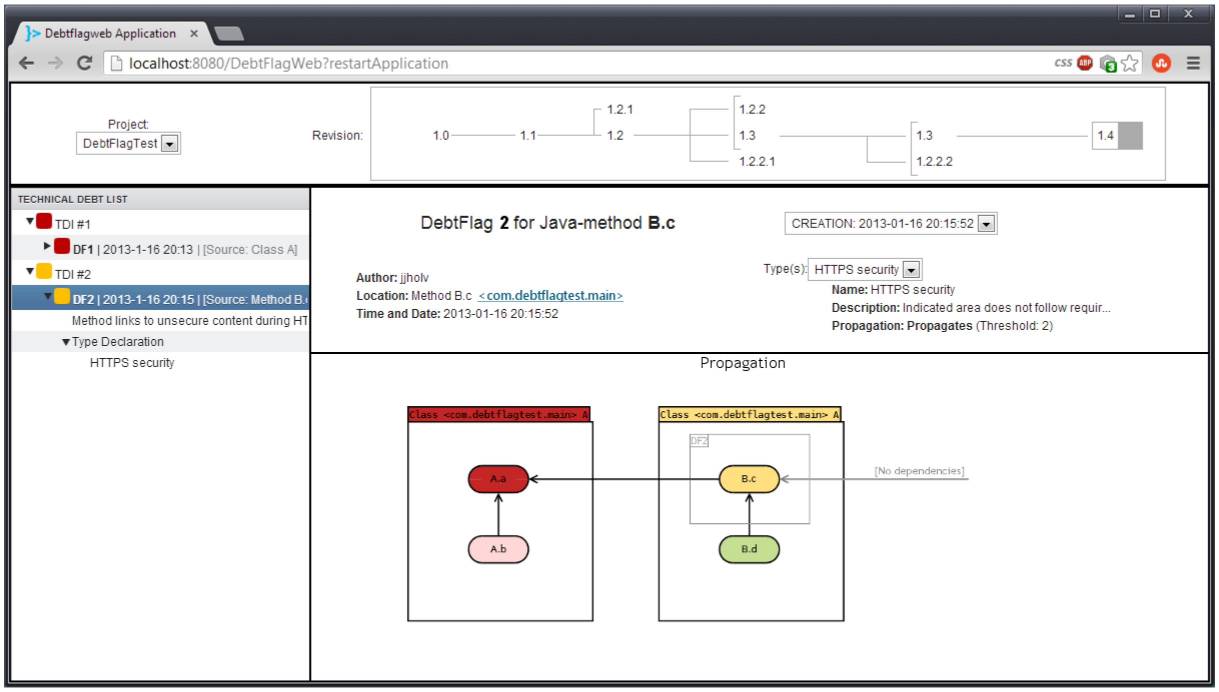
The *DebtFlag* web application has been created using the Vaadin [11] web application framework. The Vaadin data binding mechanism has allowed us to create a simple and dynamic representation of the *DebtFlag* database. This representation corresponds to the Technical Debt List.

Figure 8 depicts the *DebtFlag* web application. The header bar contains the main controls. From here it is possible to select the project and a version for which the TDL is constructed. The main content changes according to these choices and is two parted. The left hand side part contains the actual TDL. The TDL representation follows the documentation structure presented in Section II-A and it is colored according to the representation categories described in Section III-A2.

The right hand side of the main content of the *DebtFlag* web application (see Figure 8) contains detailed information for a selected *DebtFlag* element. Here the upper partition contains the attributes described in Section II-A and the lower partition contains the dependency tree. The dependency tree has the implementation elements decomposed from the *DebtFlag* element's location as its source nodes and it branches according to the rules defined by the *DebtFlag* element's types. From here, it is easy to see the reasoning for why a threshold value of a particular *DebtFlag* element was crossed.

IV. Discussion

This section covers applying the *DebtFlag* mechanism into software development. The discussion is started by establishing the mechanism's role in a software development environ-


Debtflagweb Application

localhost:8080/DebtFlagWeb?restartApplication

Project: DebtFlagTest

Revision: 1.0 1.1 1.2.1 1.2 1.2.2 1.3 1.2.2.1 1.3 1.2.2.2 1.4

TECHNICAL DEBT LIST

TDI #1

DF1 | 2013-1-16 20:13 | [Source: Class A]

TDI #2

DF2 | 2013-1-16 20:15 | [Source: Method B.

Method links to unsecure content during HT

Type Declaration

HTTPS security

DebtFlag **2** for Java-method **B.c**

CREATION: 2013-01-16 20:15:52

Author: jjholv
Location: Method B.c <com.debtflagtest.main>
Time and Date: 2013-01-16 20:15:52

Type(s): HTTPS security
Name: HTTPS security
Description: Indicated area does not follow requir...
Propagation: Propagates (Threshold: 2)

Propagation
Class <com.debtflagtest.main> A
A.a
A.b
Class <com.debtflagtest.main> A
DF2
B.c
B.d
[No dependencies]

Fig. 8. The *DebtFlag* tool's web interface

ment followed by sections discussing the expectations set for the first implementation of the mechanism. The expectations are made from a software project's perspective and cover foreseeable benefits and challenges.

A. Application in Software Development

The *DebtFlag* mechanism builds the documentation for a software product's technical debt by capturing and processing relevant observations. Capturing observations requires that the mechanism is made available in software development components where observations about the software product's state are made. These observations are processed into a TDL and automatically maintained by the mechanism. The produced technical debt documentation serves as the integration point for further technical debt management approaches and provides information to existing software development components.

In iterative and incremental software development, the implementation process relies onto previous iterations having completed their requirements as further additions and modifications are directly based onto them [12]. This makes the implementation process very sensitive to deviations in the assumed implementation state. As we have predicted this to constitute for the majority of technical debt related observations, the *DebtFlag* mechanism has been designed to integrate into the development environment; as close as possible to the developer and the process emergent to these observations.

Other software development components, emergent to technical debt related observations, are dependent onto the used software development method. The Scrum method's Sprint review [13] is an example of a software development practice the *DebtFlag* mechanism is expected to support. Here developers, who are familiar with the *DebtFlag* mechanism, take part in a process where a software product or a sub-product is assessed against currently active requirements. As deviations are found and documented, the developers give them an implementation level representation in the form of *DebtFlag* elements.

The *DebtFlag* mechanism produces and maintains a TDL according to the captured observations. The TDL can be used to integrate further evaluation and decision approaches from the TDMF. Concurrently the produced TDL provides valuable information to existing software development components. For example the Sprint planning practice of the Scrum method [13] may apply the TDL in defining new backlog items: large TDI entries may require their own backlog items, while the decomposition of new requirements into backlog entries is further defined by reviewing the amount of technical debt indicated by the TDL for this implementation area.

B. Benefits

DebtFlag captures human-made observations. In addition to relevant project stakeholders being fully aware of all active requirements and development conventions, they can provide additional reasoning for their observations. This ensures that information regarding the captured technical debt of a project is both accurate and well defined. Improvements based onto this information should be very effective.

DebtFlag documents the structure of technical debt. Software implementations are complex, hierarchical and interconnected structures. Technical debt that resides in them has similar characteristics. The *DebtFlag* mechanism captures technical debt as Technical Debt Items. TDIs are formed as a set of *DebtFlag* elements for which the *DebtFlag* mechanism automatically resolves the propagation paths. This structured form allows to track technical debt during continued development but also to apply various assessment approaches to the different levels of the acquired hierarchy.

DebtFlag presents technical debt at the implementation level. By projecting all technical debt observations onto the implementation level, the *DebtFlag* mechanism ensures that development is conducted while aware of technical debt's presence. This allows developers to avoid unintentionally increasing the value of technical debt through dependencies to affected areas or to efficiently decrease its value by resolving technical debt in areas where development is currently conducted.

DebtFlag makes continued use of higher level technical debt management approaches possible. The documentation structure is designed to be able to produce the Technical Debt List for the Technical Debt Management Framework. The extensions, that the *DebtFlag* mechanism introduces, allows to maintain this list automatically during continued development. This makes TDMF reliant management approaches applicable to the development at any given time.

C. Challenges and Limitations

DebtFlag may endorse technical debt accumulation. The documentation tools of the *DebtFlag* mechanism are designed to be as fast and intuitive to use as possible, in order to make capturing technical debt efficient enough to be justifiable. At the same time, the barrier for taking on technical debt is lowered as documenting it consumes less resources than making the optimal implementation. This is an unwanted side effect which is currently remediated by making sure the author of each *DebtFlag* element is documented. Additional measures are devised as case studies provide more information on this possible problem.

DebtFlag places the burden of technical debt management onto the end user. The *DebtFlag* mechanism relies onto the end user for identifying source points for technical debt and for resolving them at a later point in time. This indicates that the burden of technical debt management for the software product is placed onto the end users. This indicates a dependency: the state of technical debt management diminishes directly as a consequence of the end users' inability to identify or to input technical debt information into the *DebtFlag* mechanism as well as the *DebtFlag* mechanism's inability to enforce technical debt governance. In order to overcome the aforementioned problems, we intend to commit case studies to identify problems in user experience and training as well as to further develop the *DebtFlag* mechanism's ability to be cross-compatible with other technical debt identification and assessment tools.

DebtFlag does not protect the information from propagation rule set bias. Both the implementation level representation of technical debt as well as the propagation information generated for the web-interface are dependent onto the used propagation rule set (see Section II-B). While the propagation rule set does not affect the source points for technical debt, they have a large effect onto its modeled propagation and thus onto the management aspects endorsed by the *DebtFlag* mechanism. For this reason, it is important that the propagation rule set used by the *DebtFlag* mechanism is capable of reflecting the actual propagation and technical debt accumulation in the implementation. Acknowledging this, we have started a separate research on more sophisticated technical debt propagation models (see Section V-A).

DebtFlag is heavily dependent onto outside services. Unlike Automatic Static Analysis (ASA) approaches, the *DebtFlag* mechanism requires constant information about the implementation in order to function to its full capacity. While it is possible to recreate the implementation level presentation of technical debt from the database, addition and modification of *DebtFlag* elements is dependent onto having access to the development environment and implementation specific information. As the *DebtFlag* currently supports only the Eclipse IDE and the Java language, this is restrictive.

V. Future Work

We have presented the *DebtFlag* mechanism concept and the tool under development in various discussions. Attendees from both academic and industrial sectors have provided us with valuable initial feedback. Accommodating this, the *DebtFlag* tool is expected to reach its first major version during the first quarter of 2013. This will enable us to commit case studies to improve, validate and extend both the mechanism and the tool.

A. Mechanism Improvement and Validation

The current schedule will allow us to start conducting case studies with the *DebtFlag* tool during the second quarter of 2013. Here, we will first concentrate on improving the mechanism by finding solutions to the challenges and limitations presented in Section IV-C. As our department currently plays host to a variety of research where large scale software development is carried out using iterative and incremental development approaches, the first case study will be conducted in-house in order to retain the controlled environment.

This case study will be started with a thorough mapping of current product state as well as used implementation techniques and practices. This is followed by introducing the *DebtFlag* tool to the project combined with appropriate training and instruction. During continued development, we will respond to developer feedback in order to discover deficiencies in training and to enhance the user experience of the *DebtFlag* tool. Simultaneously, we will be upholding a manual identification and assessment process to gather information on technical debt and its propagation. During control periods we will be examining the differences in upholding the product's TDL with the *DebtFlag* tool and the manual process in order to discover differences between the two approaches. According to these results we will improve the *DebtFlag* mechanism.

We do not expect the aforementioned case study to solve the discussed problem of propagation modeling (see Section IV-C. Anticipating this, we have started a separate research to overcome this matter. Albeit in early-stages, our research on applying link structure algorithms, especially the PageRank algorithm [14], has provided us with promising results when used to value and indicate most crucial implementation elements for the accumulation of technical debt.

As we have intended this tool as a productivity enhancement for industrial settings, we intend to further improve and validate the *DebtFlag* mechanism in such environments. For this, we have planned and discussed a rather extensive series of case-studies to be committed with a department of a large telecommunication company. To be launched later this year, these case studies will have access to a multitude of data spanning over finished and ongoing iterative and incremental software projects. For finished products we use proven technical debt identification and assessment tools in order to simulate the life-span of technical debt. The results of various propagation models are compared against this in order to provide the *DebtFlag* mechanism with a more sophisticated propagation rule library.

For ongoing projects, we will work closely with the aforementioned party and local software development companies, in order to discover the current state of technical debt and its management for each studied software project. We will then use a refined version of our academic case-study to introduce the *DebtFlag* tool and the manual technical debt management process for these projects. During continued development, we expect to discover ways to further support the projects' technical debt management through enhancements to the *DebtFlag* mechanism. As the studied software projects will form an extensive representation of possible software development approaches, we expect that the results of these case studies will provide the *DebtFlag* mechanism and tool with adequate validation.

In committing the later case studies, we will be covering projects working on legacy software. As the *DebtFlag* mechanism is designed to capture the deviation between the current product state and its requirements, we foresee it being used to produce a mapping between legacy software components and a new requirement set. In such settings the TDL can serve as input for the modernization plan. We expect these case studies to yield additional validation for the mechanism in the form of increased legacy software development efficiency.

B. Extending the Range of Supported Techniques

After accommodating the improvements discovered by our first case study, we will extend the range of supported techniques in order to prepare the mechanism for industrial use. The plans currently encompass extending the current Eclipse plug-in to support Javadoc through the Eclipse JDT and the Python language through Eclipse pyDev, while replicating the plug-in to the Visual Studio environment in order to support the C# language.

In addition to covering a range of implementation and documentation techniques, we will be working on making the *DebtFlag* mechanism cross-compatible with other technical debt identification and assessment tools. By working together with mature technical debt identification and assessment tools (e.g. SQALE [15]) we expect to increase the accuracy and range of produced information, thus making technical debt management more robust with the *DebtFlag* mechanism (see Section IV-C).

References

[1] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, vol. 18, no. 22, 1992, pp. 29–30.

[2] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 45–48.

[3] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[4] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 23–26.

[5] M. Friedman and J. Voas, *Software assessment: reliability, safety, testability*. John Wiley & Sons, Inc., 1995.

[6] J. Kupsch and B. Miller, "Manual vs. automated vulnerability assessment: A case study," in *First International Workshop on Managing Insider Security Threats (MIST)*, 2009, pp. 83–97.

[7] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 2011.

[8] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Da Silva, A. Santos, and C. Siebra, "Tracking technical debtan exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 528–531.

[9] J. McGregor, J. Monteith, and J. Zhang, "Technical debt aggregation in ecosystems," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 27–30.

[10] Eclipse Foundation, "Eclipse integrated development environment," *URL: http://www.eclipse.org/*.

[11] M. Grönroos *et al.*, *Book of Vaadin*. Vaadin Limited, 2011.

[12] T. Gilb and G. Weinberg, *Software metrics*. Winthrop Publishers, 1977, vol. 51.

[13] K. Schwaber and M. Beedle, *Agile software development with Scrum*. Prentice Hall PTR Upper Saddle Riverˆ eNJ NJ, 2002, vol. 18.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[15] J. Letouzey, "The SQALE method for evaluating technical debt," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 31–36.