Architecting a Plug-in Based Steam Turbine Design Tool

Stefanos Zachariadis
Zuhlke Engineering
43 Whitfield Street
London, W1T 4HD
United Kingdom
zas@zuehlke.com

Tim Cianchi
Zuhlke Engineering
43 Whitfield Street
London, W1T 4HD
United Kingdom
tci@zuehlke.com

ABSTRACT

At a leading manufacturer of equipment for power generation, the engineers currently design a steam turbine, a key component of a power plant, using a large number of disjoint legacy tools written mostly in Fortran; These tools encapsulate significant engineering know how and are vital to the successful operation of the company. Their age and state pose a number of challenges, including difficulty in adapting to new methods, maintenance costs and lack of integration; the cost of replacing them all in one go however, has been deemed to be prohibitively expensive. In this experience report we describe our approach in developing a plug-in based design tool using Eclipse RCP and OSGi that encapsulates and integrates the legacy tools into a single, component-based, extendable environment that offers the advantages of an integrated solution while minimising the cost and disruption to the business and that allows for the gradual replacement of the tools. In this paper we describe the architecture and technology choices, why a plug - in based approach was used, benefits for the manufacturer and outline issues we encountered.

Keywords

OSGi, Eclipse RCP, dynamic graph, turbine engineering

1. INTRODUCTION

A steam turbine (see Figure 1) is a turbo machine that extracts thermal energy from pressurised steam and converts it into rotary motion. The latter is then used to generate electrical energy and hence steam turbines form a key component of a power station. The production of a steam turbine is preceded by careful mathematical modelling and analysis, which calculates structural characteristics based on requirements such as power output, space constraints (particularly important in case of retrofits) as well as predicts

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TOPI '11 Waikiki, Hawaii USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Figure 1: Part of a steam turbine

how the machine is going to behave under different operating regimes, such as changes in temperature, steam flow etc.

This case study describes work carried out for one of the leading global providers of power generation and transportation equipment, which has a history of manufacturing that can be traced back to the end of the 19th century. As such, over the past few decades, a number of software tools have been developed and are still used to calculate various aspects of the design and analysis of a steam turbine. Given the long history of the business, these tools are disjoint and command-line based and require manual work to create an integrated design. As such, an engineer may have to spend significant amounts of time using one tool to calculate a set of numbers, which would feed into another tool, etc. to get a complete design of the turbine and continue with the analysis. Should the analysis show an inadequately performing design, those calculations would need to be repeated. The legacy tools are also presenting issues due to their heterogeneity and age. They are written in a variety of programming languages, including Fortran and environments such as Excel, and maintaining them in order to support new designs can be costly, as modifying them may be cumbersome and fewer people have programming expertise in these languages any more.

These tools however, represent a considerable investment for the company and encapsulate complex engineering knowledge; It is acknowledged that a replacement with a modern

integrated tool is necessary; Such a tool would offer productivity boosts, since the engineers would not need to work in disparate tools, would be more maintainable and could offer more modern features (high level validation, visualisation, etc.). The cost of replacing them in one go, however has been deemed to be prohibitive given their pervasiveness. In this paper we describe the Turbine Design Tool (TDT), a plug-in oriented approach that we are using to integrate and gradually replace these tools. This approach is plug-in based in two ways: Firstly, it is built on top of OSGi as an Eclipse RCP application; more interestingly, however, we describe an approach where we encapsulate the existing tools as individual plug-in components, which are transitively composed by our framework automatically in order to design and analyse a steam turbine in an integrated tool. For reasons of brevity and confidentiality, this report cannot not include all details of the architecture, but will rather focus on the plug-in nature of TDT.

2. ARCHITECTURE

2.1 Motivation

The design and architecture of TDT described in this section was motivated by the desire to replace the existing tools with a more modern approach that would be integrated, offer improved usability and performance and be more maintainable, while minimising the cost and disruption to the business. As such, we developed a framework that allows for the gradual replacement of the existing tools, while being extensible as to rapidly support emerging developments in mechanical engineering and offering the benefits of an integrated solution.

2.2 Calculators

The foundation of TDT is the Turbine Domain Model, or TDM, which is a model that represents a steam turbine. The model allows for an object representation of structural parts of the complete turbine, such as modules, blades, etc. along with any attributes or *parameters* that may be attached to them, such as lengths, pressures, etc. As such, when designing and analysing a turbine in TDT, we are interested in creating an instance of the model and manipulating its attributes.

We decided to break down the computation required to design and analyse a complete turbine into a set of discrete *calculators*. A calculator is defined as a black box that requires a well defined set of inputs and produces a well defined set of outputs. The set of inputs and outputs map to values in the Turbine Domain Model. Calculators are technology - independent; hence the existing legacy tools were encapsulated into individual calculators, a process that involved significant business analysis and requirements engineering, to enable mapping the inputs and outputs of the calculator to the TDM.

Since the legacy tools predate TDT and the TDM, often by decades, calculators are domain model agnostic. We define a *path* as a textual representation that represents a parameter in the TDM. The path can be considered as a *pointer* to the parameter in the TDM as it does not include its value. As such, a calculator defines its inputs and outputs in terms of paths. It accepts and produces input and output in terms

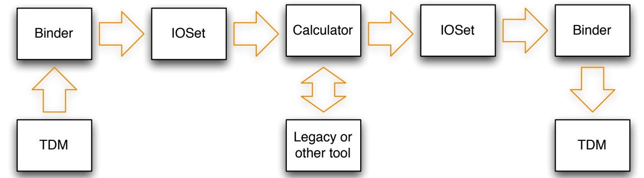

Figure 3: The interaction between calculators, IOSets and the TDM.

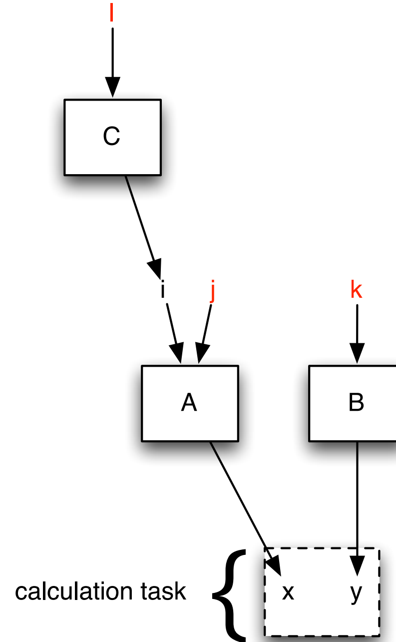

Figure 4: An example of chaining. Boxes represent calculators, lower case letters represent parameters.

of key - value pairs called *IOSets*. The key of an IOSet is the path to a parameter and the value is the number or string that the path points to in the TDM. Consequently, as a calculator interfaces only via IOSets, it is independent of the TDM and can encapsulate legacy tools. We use a component called *binder*, to map to and from the TDM and an IOSet. Figure 3 shows the interaction between the TDM, the binder and an individual calculator.

2.3 Tasks and Chaining

Engineers require to work on and refine individual parts of the turbine, for example the moving or fixed blade rows. We define a *calculation task* as collection of parameters that represent the underlying task and need to be computed. TDT offers a number of design (e.g. what should the width of a blade be?) and analysis (e.g. how does the turbine behave under varying pressure?) tasks. When an engineer selects a task to work on, TDT uses what we call a *chaining framework* to dynamically determine the calculators that need to run.

The chaining work as follows:

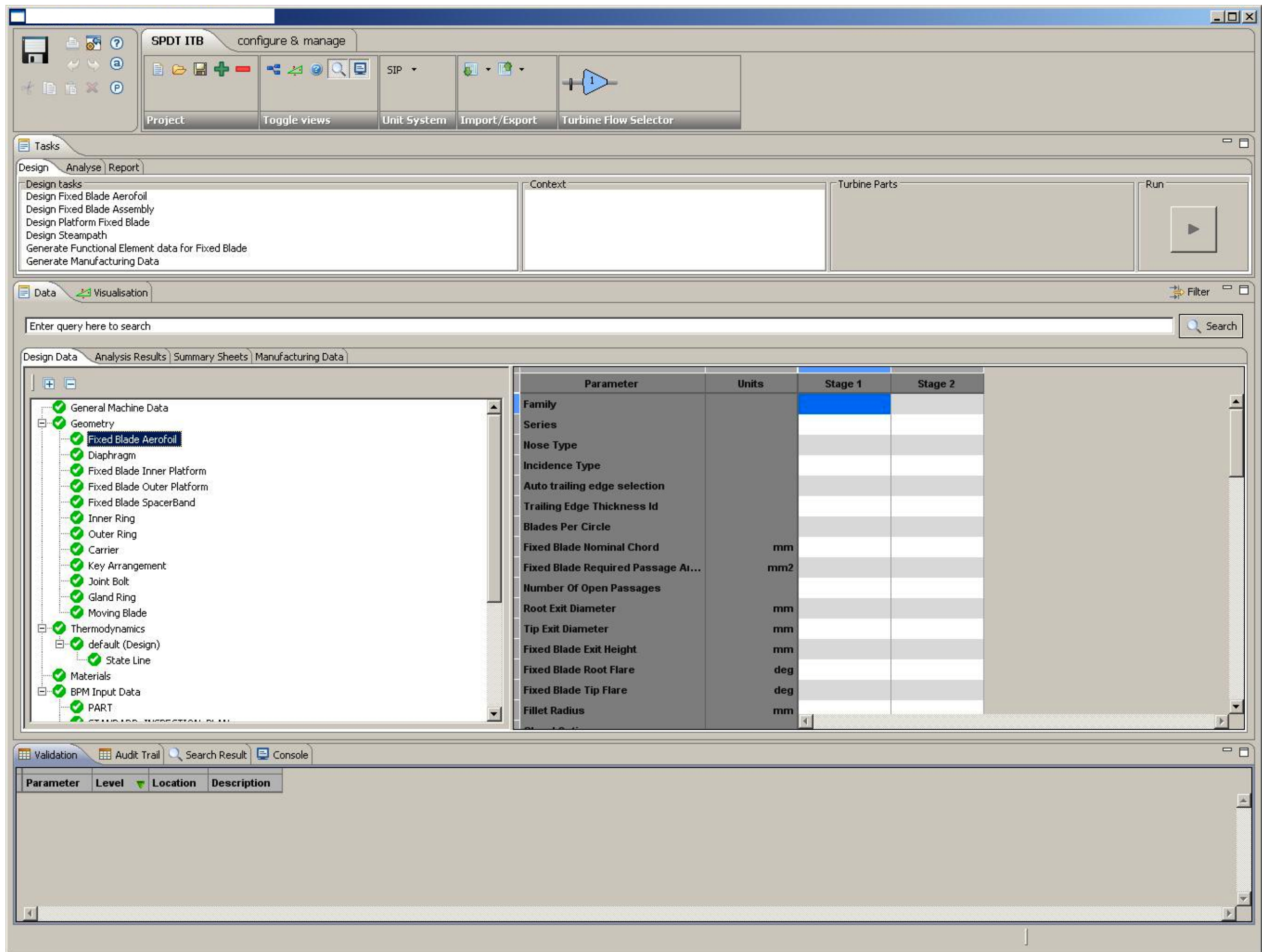

Figure 2: TDT running, showing the calculation tasks and some parameters.

1. A user selects a calculation task to be executed
2. The task is defined as a collection of parameters
3. TDT goes through all the parameters and for each of them checks if one of the available calculators can produce it
    - If no calculator is able to produce the value, then the parameter becomes a user input
    - If a calculator is found, then for each of its inputs:
        - if the TDM contains a value for that parameter, then the value is added to the graph
        - if not, then process is recursively repeated

The framework is inspired by the transitive dependency frameworks found in tools such as the Debian package management system[3] or the Maven dependency management approach[4]; similarly, TDT calculators describe artifacts (parameters) that they consume (their inputs) and artifacts that they produce (their outputs). This is graphically demonstrated in Figure 4. The user chooses to run a calculation task that prescribes that parameters x and y are to be computed. TDT determines that parameter x can be computed by calculator A. Calculator A requires parameters i and j as inputs. TDT determines that i does not exist in the TDM but can be calculated by calculator C. Calculator C requires parameter l. Parameter y can be calculated by calculator B, which requires parameter k. No calculators can compute j, k and l and if values for those do not exist in the TDM they will become inputs and will be requested by the user prior to execution.

TDT therefore transitively determines all the calculators that need to execute and all the values that need to be in place in order for the parameters determined by the selected calculation task to be computed. The result of this process is a directed graph[1] or *chain* of calculators and parameters, with the leafs of the graph representing the parameters that the calculation task encapsulates, and a set of values that need to be inputed manually by the user. It is assumed that only *one* calculator can calculate a particular value at any time. This assumption can be validated when TDT starts.

Engineers are only exposed to tasks directly and not to the individual calculators. Calculators are therefore effec-

[1]The graph produced may be cyclical or acyclical, based on the underlying physics. We have mechanisms in place that handle cycles by checking for oscillations and convergence, but these are considered to be out of scope for this paper

tively plug-ins that are graphed together by TDT to calculate various aspect of the turbine including the full design and analysis. When executing a task, TDT uses the binder to give the values that are required by every calculator and to populate the TDM with the results of the calculation.

This approach has various benefits for the company. Most importantly, by breaking down the computation into individual plug-in calculators, TDT can reuse the legacy tools, while being able to provide most of the benefits of a modern integrated tool. Moreover, since TDT considers calculators as black boxes and they can be tested individually, the approach allows for their gradual and individual replacement, while keeping the overall functionality of TDT intact. Finally, as new engineering technologies and methodologies are researched and become available, the TDM can be suitably extended via further modelling and new calculators can be built that encapsulate the new business logic and reused by the framework with relative ease.

2.4 Eclipse RCP and Other Implementation Choices

Figure 2 shows a picture of TDT running. We have chosen to build TDT on top of OSGi[5] and the Eclipse RCP platform[2]. Figure 2.4 shows a high level over view of the software architecture. The UI is built using a set of Eclipse plug-ins using SWT, JFace and other libraries. It communicates with the rest of TDT via a *domain controller*, an implementation of the front controller pattern[1]. The domain controller abstracts away from the calculators, the calculation tasks that are available, the TDM and the various services such as validation that TDT provides. All are also offered as Eclipse plug-ins.

The plug-in architecture of OSGi / Eclipse RCP fits naturally with the modular approach we have chosen via the calculators and the chaining framework, as the framework is calculator - agnostic and operates on sets of plug-ins (the calculators) that can change over time. Moreover the rich versioning metadata provided by OSGi can be used to deterministically denote which version of all TDT plugins was used to work on a design; this has particular significance for a safety - critical manufacturer, the products of which such as a steam turbine are in production over a very long period.

Finally, the Eclipse update mechanism is used to deliver the application and updates to the end users; The company employs a large number of engineers working all over the world and can benefit from the ability of the update mechanism to deliver updates to individual plug-ins instead of monolithically re-delivering the complete TDT if any change is made to the tool.

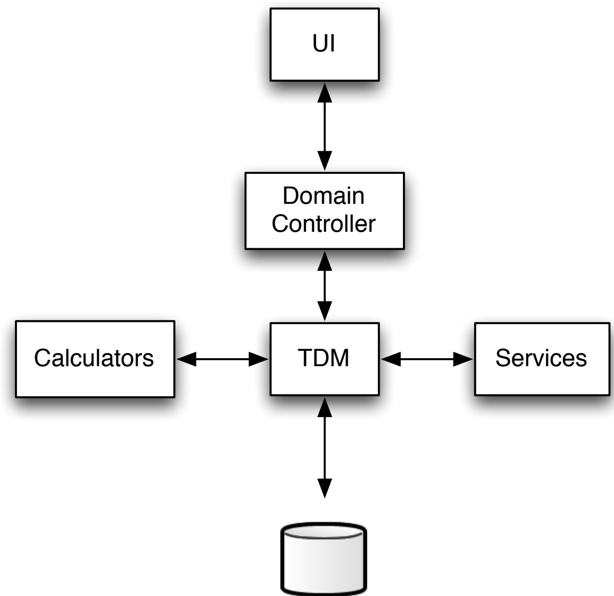

Figure 5: The base TDT architecture.

3. CONCLUSIONS

TDT is currently under development, though test releases are becoming regularly available to key users. The initial feedback that we are getting is highly encouraging as we are able to deliver the integrated tool rapidly and our users are reporting significant increases in efficiency and speed, due to the nature of the integrated development environment. We are also able to provide a number of advanced new features, such as validation, consistency checking and more. The chaining mechanism also offers further opportunities for improvements in the process, such as parallelising the execution of calculators, to take advantage of modern multi-core hardware.

We note that while the approach is proven to be technically sound as it begins to get used by our users and offers significant advantages as outlined above, it also poses a number of challenges; most importantly, engineering a domain model that is expressive enough to represent the whole turbine, and mapping the inputs and outputs of legacy tools to that model is where TDT is spending most of its effort on. We anticipate a productive release this year, followed by the gradual replacement of some of the underlying tools. We have found that the plug-in based approach of wrapping the legacy tools offers a pragmatic mechanism that delivers a modern tool quickly and with minimal disruption.

We believe that the approach outlined in this paper can be generalised to other software integration projects, where individual disjoined tools that are used as components to achieve a result are composed into an integrated product. This would require: i) creating a domain model to represent the end result ii) wrapping the tools iii) mapping their inputs and outputs into that domain model and iv) using a graphing algorithm to dynamically compose them.

4. REFERENCES

[1] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison Wesley, Reading, Massachusetts, Nov. 2002.

[2] A. Kornstadt and E. Reiswich. Composing systems with Eclipse rich client platform plug-ins. *IEEE Software*, 27(6):78–81, Nov./Dec. 2010.

[3] M. F. Krafft. *The Debian system: concepts and techniques.* No Starch Press, pub-NO-STARCH:adr, 2005.

[4] Maven. http://maven.apache.org.

[5] The OSGi Alliance. OSGi service platform — core specification, Aug. 2005. Release 4.