An Introduction to JeeWiz for Software Development Managers

Matthew Fowler

Product Manager and Chief Architect - *JeeWiz!*

Introduction

JeeWiz can automate any aspect of system development activity that is repetitive. The vast majority of the programming, configuration and deployment activities in system development are repetitive, so JeeWiz can automate them.

The starting point for JeeWiz is a high-level specification of what needs to be built - a UML model or its equivalent in XML or some other format. Using patterns that describe the conversion to target systems, JeeWiz converts the specification into a complete 'do-nothing' system. An application programmer can then add the parts that are not repetitive using standard development tools such as IDEs (integrated development environments) or text editors.

The key characteristics of JeeWiz are:

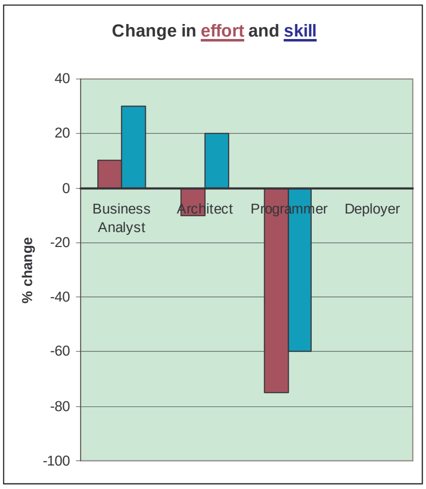
- the range and complexity of the automation patterns. In typical business applications, 75 to 95% of the development process can be automated, including all the connective code and configuration
- the integration of these patterns across all the development products - source code, web pages, deployment information and build jobs
- the ability to encapsulate what is not repetitive (the "business logic"), so that the specification, architecture and business logic can change independently.

JeeWiz includes very fast rapid deployment environments for Java/J2EE and C#/.NET. However, JeeWiz is more than just a 'RAD' (Rapid Application Development) environment: the generation can be customised to match a project's technical architecture. JeeWiz is also universal: it can automate other languages and deployment environments.

This paper describes JeeWiz from a software development manager's perspective. Companion papers describe JeeWiz from other viewpoints.

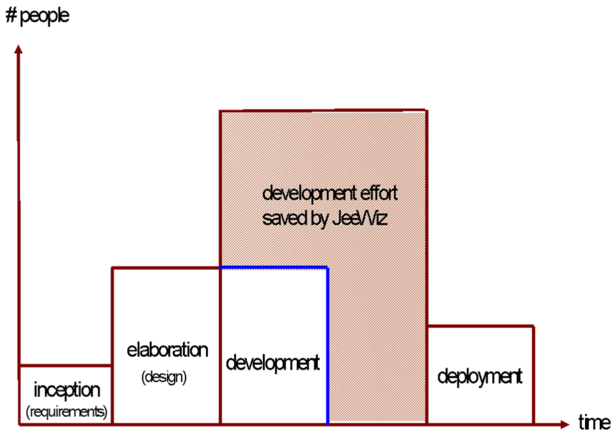
Effort and Skill

The impacts of using JeeWiz on the main project roles are as follows:



- Business analysts have an increase in work and skill required, to add extra detail to the specification.
- Architects have less to do because the coding patterns and standards can be implemented by varying the JeeWiz transforms, but this requires some JeeWiz skill.
- Programmers have significantly less to do because most of the technical (infrastructure code) is generated. The remaining code is the business logic, and this is simpler because the infrastructure has been removed - so less skilled programmers can still be effective.
- There is minimal change for deployers.

The graph below shows how the classical cost/time graph for a project is altered by using JeeWiz:



JeeWiz typically saves at least 75% of the effort in the development phase, including testing. As the development phase consumes about 60% of the total cost of new developments, the JeeWiz saving is 45% of the total project cost.

There will also be elapsed time saving, although less dramatic than the cost savings.

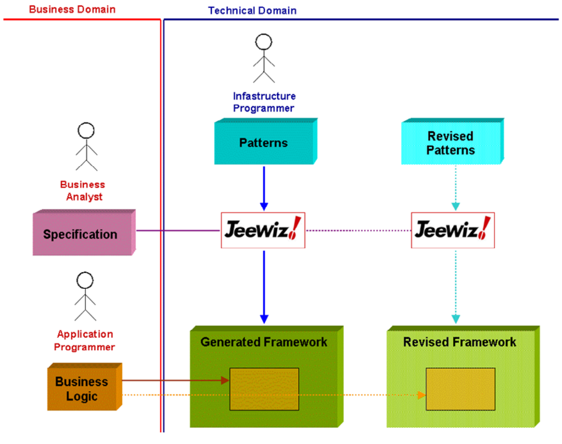
This level of savings is available now with the J2EE and .NET environments. Because JeeWiz is universal, it can address other environments, as demand dictates.

The Agile Architecture

JeeWiz separates business and technical concerns. The business analyst and the application programmer do not need to know or use any aspect of the target platform.

The business analyst creates a logical specification: he models 'entities' rather than the mapping of an entity onto a platform like J2EE (remote and local interfaces etc.).

The application programmer manually codes the business logic in a separate set of files using Java or C# and his favourite IDE. The code to be written is defined by the specification and headers are generated (and maintained) by JeeWiz. Dependencies on services are wrapped up in platform-independent proxies.



It is the job of the architect to describe how to turn the specification into a particular architecture; and this is done independent of the specification of the application being deployed. This means that the business and technical domains are quite separate, which has massive benefits for project management:

- There is no critical path dependency on the architecture: business analysis and application programming can proceed independently.
- Late changes in the architecture can be accommodated easily.
- It is possible to deploy the same specification and business logic to two different platforms
- Platform changes in the maintenance phase are easily accommodated.

Should a product development company want to deploy the same system on .NET and J2EE, there is even a way of writing the business logic so it turns into C# or Java.

Separating the technical and business concerns with JeeWiz makes the business analyst and infrastructure more valuable to the enterprise, because their output feeds directly into deliverable systems (rather than documentation of standard patterns, for example).

Applications

As mentioned above, JeeWiz provides rapid development environments for building web-based J2EE and .NET information systems. The standard patterns build a tiered system by default: presentation (web pages); business operations ; data views (for screen display across multiple entities); and data (entities and relationships). Variations on this theme are:

1. JeeWiz is usable with a minimum of 'specification' - at an early stage of modelling. Very few values are required; where optional values are omitted, JeeWiz inserts sensible defaults. This means it can be used for very early proofs of concept and verification of requirements with users. As the project proceeds, the specification can be incrementally elaborated to become a detailed system specification.
2. The patterns used in JeeWiz can automatically 'project' one tier into another. For example, a new attribute on an entity could be projected into data views in the business operation tier and from there onto screen pages - which are smart enough to lay out the new information appropriately. This reduces the maintenance cost when the new information can be part of the specification. Often, the business logic will need to be changed as well; this will also be easier to maintain because it is smaller and simpler.

   This will expand the viability of packaged products, because the cost and time of customisation falls dramatically: instead of the last 20% of functionality costing 80% of the budget, it would be nearer to 10-20%.
3. Another way JeeWiz can be used is to generate integration software - to connect from one tier of the architecture to other types systems. New patterns can be developed to integrate with these new systems: JeeWiz will automatically maintain the integration code based on the existing specification. Most patterns of this type only take a small number of man-days to complete, so if there are more than four or five calls to be made across an interface, the JeeWiz pattern will save time overall.
4. The generated system can be reduced in scope just by changing configuration values. For example, JeeWiz has been used to create EJBs behind BEA's WebLogic Integration in a Web Services application, omitting the presentation tier.

Alternatively, the patterns could be adapted to output specialised code (e.g. including relevant tags for inclusion of code in WebLogic Workshop).

JeeWiz has even been used to generate other technical products, such as XSL-T transforms (which are otherwise horribly difficult to maintain), XML Schemas, HTML documentation and some of JeeWiz itself.

Long-term we expect JeeWiz to support a new kind of programming, where the goal is to discover patterns in the development process and then automate them. This involves creating the sort of descriptions discussed below: describing the allowed specification; and then writing the programs to automate. Because JeeWiz allows the descriptions to be re-used in other environments, this development only needs to be done once.

This approach is particularly relevant to the integration of software systems. Where the systems are similar, the integration can be completely automated using a specification of one side to generate the required software.

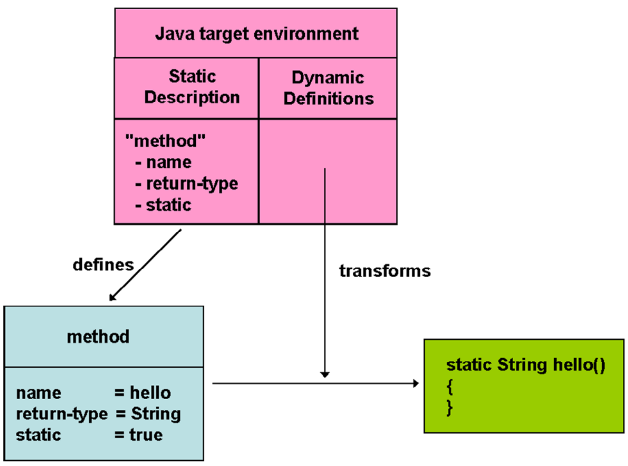
How it works

Many people have trouble believing JeeWiz is possible. This section gives an overview of how JeeWiz works, to demonstrate informally that it is possible!

Describing Specifications

The first key innovation in JeeWiz is that it provides a modular, customisable way of describing target environments, such as languages (like Java or C#), modelling abstractions (like UML) or business information systems (such as J2EE or .NET). These descriptions cover

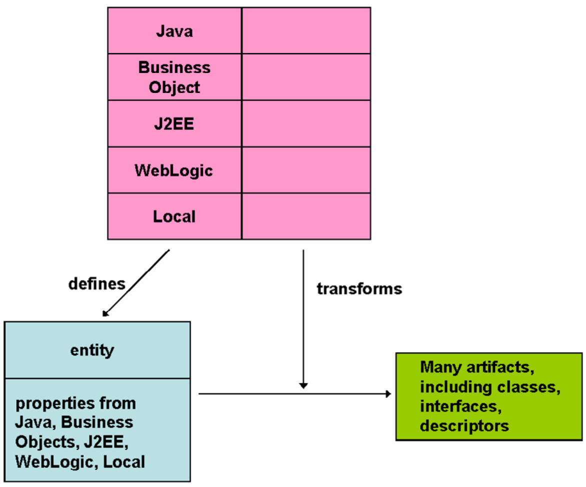
- what can be specified - the format of a specification - in the target platform, such as whether a method is static, or whether a class is an entity or session (control)
- how to convert the specification into deployable components, such as creating a Java program or an entity EJB.



The motivation for having a description of the target environment is that

- the specification can use concepts of the target domain
- the conversion of the specification into a delivered system is flexible, and can realistically be customised for particular customer environment or even complete new technologies.

Now, this doesn't address the reality of most systems, which have multiple target environments working together. For example, a WebLogic system has concepts from at least four technologies - Java; business object and entity-relationship abstractions; J2EE; and WebLogic-specific features. The modularity feature of JeeWiz make it possible to describe a real environment as a group of target environments, but have a single specification covering them all.



This diagram shows the use of a group of descriptions (Java, ... local) which define the concepts available in the specification - here we have an entity. In specifying the entity, we can use properties from any of the descriptions, such as the name in Java and remote accessibility from the business object layer.

This construction is flexible: we can substitute the .NET description for the J2EE/Weblogic descriptions. In this way, we re-use the Java, Business Object and local descriptions - many of the concepts, properties and transformations are the same.

The crucial point about using specifications in this way is that the higher-level concepts are much more expressive. For the same effort, they magnify the information content of the specification. For example, a Java specification is roughly the same work as writing the code, but a J2EE specification produces about 20 times more artifacts.

This approach to describing systems is modular and customisable. This means that new technologies and local variations in patterns can be incorporated in exactly the same way - by creating a new description level. The specification language can be devised to reflect the standard analysis language. In the early stages, the specification can be developed by a business analyst as a way of recording classes, operations and methods. Later in the development cycle, a technical analyst may add more detail.

The specification becomes the hand-over point between the analysts and the technical team: it is a higher-level bridge between business and IT. We emphasise that this is not an English-language specification, with all the inexactness that entails: a JeeWiz specification is precise. It must conform to the rules laid down in the description.

We use the term 'specification' rather than 'model'. There are a number of reasons for this, but the primary one is to emphasise the level of detail and exactness that can go into the specification. In the early stages, JeeWiz uses UML models as its input. But then JeeWiz also allows new, higher-level constructs and more detail to be added in the UML tool, to create a complete specification. Most development groups today only use models to facilitate communication between stakeholders in the development process; once a technical specification has been deduced and agreed, the model tends to be forgotten. In contrast, the JeeWiz specification builds on the modelling effort, adding the detail to become the contractual interface between business and technical. JeeWiz uses the result to determine the structure of the delivered system.

JeeWiz supports a number of UML tools - Rational Rose, System Architect, Magic Draw, Visio. Rational Rose and System Architect allow plug-ins, which give GUI support for the additional specification items in JeeWiz. In tools without a plug-in capability, the target-specific vocabulary is entered using UML stereotypes and tagged values.

In addition, the specification can be written in XML (this is the native format) or derived from Javadoc tags for import from Together-J models.

In the early days of project elaboration, a proof of concept may be built from a minimal specification - simply define a few classes and attributes, and let JeeWiz fill in the blanks to build a system with the structure of the eventual system but minimal business logic. As elaboration proceeds, more and more detail can be added to the model/specification to cover datatypes, constraints, relationships and so on, to completely specify a final system.

There is therefore more effort involved in creating the specification than is usual in building a 'model'. However, against this must be set the cost of writing detailed technical specifications, which becomes unnecessary, and the effort to build the system - most of which JeeWiz can do.

Generating Systems

While the ability to specify systems is necessary for JeeWiz to operate, the really exciting part about JeeWiz is that the generation side can codify the usage of the target environments.

This means that we codify, and then automate, any aspect of building the target system that is repetitive. This applies to much of the programming, documentation and testing tasks.

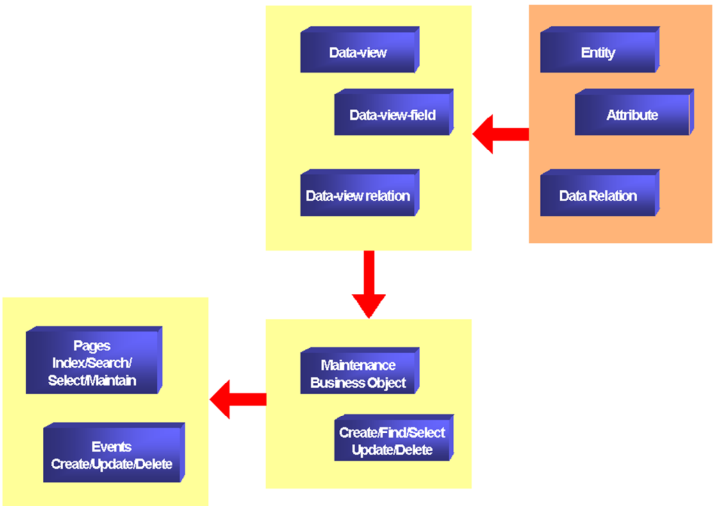
How is this possible in JeeWiz when so many other attempts to achieve this goal have failed?

The main issue is the huge scope of the transformations required of automatic generation systems. One part of a transform may address a tiny detail - how to construct variable names according to company standards - while the overall goal of the transform is to solve a high-level policy issue, such as whether to give users access to business methods on entity beans. Size and complexity is also an issue: JeeWiz version 2 had hundreds of transforms, and it just addressed J2EE back-end systems.

The solution is to take large-system development techniques and apply them to the generation process. JeeWiz uses a wide range of these techniques, but the main ones are

- hierarchical composition (re-use of other transforms)
- object-oriented approach to modularity (generation patterns attached to the data specification)
- polymorphism (overloading of transforms to create specialised functionality)
- automatic application of patterns (like event-handler in GUIs).

The highest level of these transforms are 'mega-patterns', which assemble whole groups of other transforms to provide very high-level functionality. The following example shows three mega-patterns, that convert specifications of entities into maintenance session beans, aggregation objects and finally web pages (using JSP or ASP.NET).



Although mega-patterns are the most powerful patterns, they are the easiest to write. The mega-patterns shown above completely automate the generation of the data maintenance part of an application are only a page long.

Individual transforms are organised using the specification descriptions described above. The complete group of transforms is assembled modularly depending on the configuration being built. This modularity, and the ability to re-use transfoms from lower layers, makes it feasible for architects to adapt the transforms for local requirements.

This means that the same specification can be used to generate systems on different platform, such as .NET or J2EE. It also means that, as new technology standards are introduced, it is possible to attach them in the appropriate place in the architecture. For example, JDO (Java Data Objects) is incorporated as an alternative to entity EJBs, but still running within the J2EE container.

The owner of the transformations is the architect. During the inception or elaboration stage, it often possible to use the JeeWiz standard patterns without significant modification. However, JeeWiz is designed to be customised and extended, and we expect many architects will do this. Very simple customisations are to change names and directory structures to match corporate standards, or to adapt the JeeWiz build to a larger system. More ambitious customers have reworked entire layers to support obsolete versions of the EJB standard, involving writing a large number of patterns.

Incorporating "Business Logic"

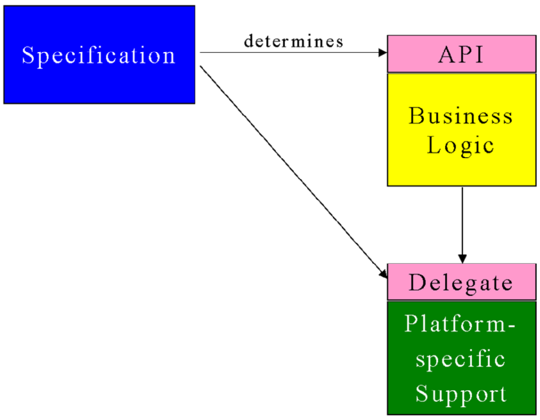
It is important to realise that the techniques described above don't just generate a code outline: they generate complete systems incorporating links to the platform, bespoke local infrastructure, constraints, logging and tracing, deployment descriptors, build jobs and so on.

Screen pages and their navigation can be specified, as well as the aggregation data for each use case. JeeWiz turns these into linked pages automatically (as shown in the standard demonstration files).

An important part of this process is to generate the "connective code" where appropriate. For example, a screen page may invoke a business method, passing its data as a parameter; the standard create/read/update/delete business method will invoke a method on an entity and return an object to the caller; which will display the result on another page. All this connective code is also automatically generated. The patterns are written in Java or C# syntax. Where variations in the language or environment occur, the code is generated by a pluggable 'language control'.

This is a far cry from the 'code generation' features provided by modelling tools today, which may generate a skeleton of 2 or 3% of the code (and none of the other artifacts) for a standard information system application. JeeWiz generates 90-95% of the code and other artifacts for this type of system, including web pages using HTML, JSP or ASP.NET, Visual Studio projects for .NET or Ant build projects for J2EE, deployment descriptors and configuration files.

So what is left? This is what we refer to as the "business logic" - but it is actually any code that isn't currently covered by a pattern. This must be developed by an application programmer in an appropriate IDE. JeeWiz generates an application framework which is a complete application less the business logic; the application programmer fills in business methods and sometimes screen logic (although ideally not often), as prescribed in the specification and reflected in the framework.



The application programmer must implement the API, which is determined by the specification.

JeeWiz hides the details of the implementation platform and technology from the application programmer behind a delegate. This is also determined by the specification and generated by JeeWiz. It provides the service need by the programmer without exposing the platform.

This not only makes it easier for the programmer to write the business logic - he doesn't need to know the technical details - but potentially the same source can be used on J2EE and .NET, or migrated more easily to some other platform.

The business logic goes into a separate directory tree. The directory tree contains class files that need to be implemented by hand - and nothing else. This makes it easy for the application programmer to see what must be done.

The end result is that in typical application environments, there will be fewer application programmers needing less technical skill. They will develop into business experts - making them more valuable in the eyes of the business.

Deployment

Because JeeWiz offers the possibility of mapping to multiple platforms, JeeWiz supports (and recommends) that the original specification be restricted to logical items, i.e. those that are not target-specific.

Things like database table-names may have different configuration names in targets, but they are considered logical - and can be specified in the original specification - because they will be the same value in any database-oriented deployment.

Other information that has no similar concept in other platfoms can be added in a number of ways:

- By simply using a default - this works in a surprising number of cases. This default can be the value specified in the distributed transforms, or a special value added by the architect in an override of the transform.
- By specifying a sensible default as a function of other information in the logical specification. For example, table-names can be derived from entity names
- By embellishing the JeeWiz XML specification.

The approach of adding information to the JeeWiz XML specification raises the issue of how to keep the original specification in-step with the embellished physical specification. JeeWiz has a merge utility for this situation. It means that specific deployment information can be added to a build, but that the logical information from the original specification takes precedence.

Another benefit of restricting the original specification to logical items is that multiple deployments of the same specification can be made on "different" platforms - which could be different technologies or upgraded platforms as the application is maintained.

Conclusion

JeeWiz can automate any area of system development that is repetitive. The standard product provides highly automated environments for J2EE and .NET development, plus features for similarly automating and customising related or completely different environments. Automation patterns need only be written once and can be re-used or customised. JeeWiz provides the ability to describe new technical areas and automatically convert specifications to systems: it is a new kind of programming.

About the Author

Matthew has a degree in Computer Science from MIT. He spent 5 years of his early career creating new computer languages and application generators, and building application systems with them. He has since held a range of technical, product management and business roles in LANs, communications, software engineering, software cost estimation and enterprise systems. Since 1999 he has been struggling with the architectural, technical and managerial issues of J2EE. This led to the concept for JeeWiz, an MDA product, for which he is product manager and architect.