

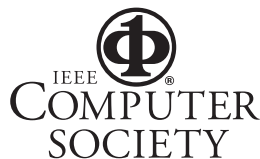


Agile Programming: Design to Accommodate Change

Dave Thomas

Vol. 22, No. 3
May/June 2005

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.



© 2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/portal/pages/about/documentation/copyright/polilink.html.

Agile Programming: Design to Accommodate Change

Dave Thomas

Agile development provides a set of practices simple enough to engage developers, managers, and customers yet sufficiently sound and disciplined to build quality software with predictability (see www.agilemanifesto.org).

Applications are expected to evolve over time as their requirements change, and agile development's refactoring and testing practices accommodate software evolution. (Indeed, Extreme Programming's maxim is "embrace change," and today's best developers view refactoring as a badge of honor.)



Refactoring improves code, usually increasing the function while reducing code bulk.

However, such refactoring or restructuring often forces the application to undergo a complete development cycle, including unit, acceptance, and regression testing, followed by subsequent redeployment. In a large IT or engineering production system, this can be time consuming and error prone.

Agile programming is design for change, without refactoring and rebuilding. Its objective is to design programs that are receptive to, indeed expect, change. Ideally, agile programming lets changes be applied in a simple, localized way to avoid or substantially reduce major refactorings, retesting, and system builds.

Table-driven programming encompasses four well-known but perhaps forgotten agile-

programming techniques that help anticipate and accommodate many common changes.

Table-driven programming

In the early 1970s, systems programmers took great pride in using table-driven programming, much as programmers today might talk about their models or design patterns. It's a proven technique for implementing policy-driven systems, and it uses state pattern matching, concurrency, and workflow; decision tables (business and engineering rules); and constraints (spreadsheets).

Separating policy from mechanism

The principle of separating *policy* (what) from *mechanism* (how) is a best practice in design. A policy-driven system uses a set of rules to describe a specific policy and an associated specialized policy "language" runtime that provides the execution mechanism.

Given a well-designed policy language, an analyst, engineer, or end user can often completely describe the system behavior. This eliminates the potentially expensive and error-prone step of translating from business requirement to implementation.

State tables

Most designers are familiar with simple state tables, state transition diagrams, or regular expressions. The difficulties with maintaining huge state machines found in large complex mission-critical applications led to the

breakthrough development of state charts,¹ which brought modularity to state tables. The combination of state charts and objects produced several powerful variations called *object charts*. UML provides object charts, which are unfortunately a little more complex than many applications need.

State tables are most commonly used for validating input sequences, for simple parsing and event processing in user interfaces,² and for managing asynchronous concurrent communications events. Unfortunately, far too many user interface designers fail to appreciate the state space's complexity, hard coding the state model rather than designing and implementing a proper state transition model. State tables are used in engineering for process control and in business for complex workflow.

Decision tables

Few developers are familiar with decision tables—one of the simplest and most powerful techniques for dealing with complex logic. A decision table is unique in that an end user can easily specify and maintain it. The table comprises a set of conditions placed above a set of actions to perform (see Table 1).

For each combination of conditions (c1, c2, ...), a rule exists (r1, r2, and so forth). Each rule comprises a Yes, No, or Don't Care (-) response and contains an associated list of actions (a1, a2, and so forth). Then, for each action, an action sequence number specifies the order in which an action should be performed if this set of conditions is true. For example, if c1, c3, c4, and c5 are all true, then a1 should be followed by a3.

This simple table format makes it easy for end users to specify the conditions, actions, and rules. Users needn't worry about complex nested *if then else* statements or be confused by *and/or* when describing complex logic. Additional tables can be automatically checked to determine if rules are redundant or ambiguous.³ Finally, an *else* rule either traps all unspecified cases, or the cases trigger an exception. Designers can decompose very complex logic into multiple tables, which invoke other tables as actions.

I've applied decision tables in do-

Table 1
A decision table, comprising a set of conditions and a set of actions to be performed

Condition stub	Conditions/actions	Rules				Else
		r1	r2	r3	r4	
	c1	Y	N	Y	N	-
	c2	-	Y	-	N	-
	c3	Y	N	N	N	-
	c4	Y	-	N	N	-
	c5	Y	Y	Y	Y	-
Action stub	a1	1	2	-	3	-
	a2	-	1	-	1	2
	a3	2	3	2	-	-
	a4	-	-	1	2	1

mains from embedded real-time systems to library loans, and I'm always amazed that the users can diagnose the problems and often repair them without developer intervention! David Parnas has discussed the use of tableaux of a similar nature in formal documentation for mission-critical systems such as nuclear reactors.⁴

Spreadsheets

Many policies exist that users can conveniently express in a simple spreadsheet using their favorite desktop tool. These policies can be used daily to define everything from salaries and benefits to drill-holes for engine blocks. This makes spreadsheets a natural tool for defining policies. Executing them using a spreadsheet tool is straightforward, provided you apply some discipline—such as remaining consistent with the naming and layout.

Few developers are familiar with decision tables—one of the simplest and most powerful techniques for dealing with complex logic.

Implementation

A generator can help automatically translate policies into Java, C#, or C++, as is currently done for UML state and message sequence charts. However, for many years, developers have been translating policies into tabular data structures and then evaluating them using a simple interpreter.

This latter style of table-driven programming has important advantages. First, because the policy is represented as data, developers and sometimes end users can change it on the fly and use it to support mission-critical nonstop applications. Second, the tables and interpreter require substantially less space than the compiled representation. The cost of interpretation is minimal for 90 percent of applications whose performance is dominated by other factors.

Clearly, being able to test and manage tables is important. Developers can apply user-centered testing tools such as FitNesse (<http://fitnesse.org/FitNesse>. WhatIsFitNesse) to the tables, and the tables can be stored in the configuration management system and managed along with the application code. Furthermore, because the tables are represented as data, you can easily modify them using maintenance transactions or table editors and deploy them in the application as soon as the next transaction.

The ability to dynamically reconfigure applications will become increasingly important as businesses evolve into real-time distributed enterprises. There's no more agile business than

one that can change its business policies, test, and redeploy without recompilation or reloading.

State tables have obvious implementations as a 2D array, which can be represented as a sparse array for very large tables. Object charts can have subtle semantics, so I generally recommend using a well-designed evaluator.¹

In the case of decision tables, my favorite implementation is the *rule mask technique*,³ a compact and efficient interpreter. Tables are represented compactly as bit vectors and a simple interpreter that uses bit operations. This implementation also provides a simple technique for definition time checking of conflicts or ambiguities.

Spreadsheets can be parsed with a simple parser that records the dependencies such as *c11 needs a11 and b11 to compute a11 + b11*. The simple constraint system is easily solved by sorting the dependencies using a topological

sort.⁵ Clearly, if special formulas or macros exist, you need to provide the equivalent action code. To evaluate the spreadsheet, you walk the sorted dependency table in order, evaluating each element.

From the outset, agile development focuses on accommodating program evolution. It's important in design to consider those points of the system that will likely undergo substantial change. At such points, it's often appropriate to apply agile-programming techniques.

Software designed using older table-driven techniques is often far more portable and malleable than equivalent human- or machine-generated code. Because it's much easier to change data than code, data table representations allow programs to be changed on the fly, often by end users! ☺

References

1. M. Samek, *Practical Statecharts in C/C++*, CMP Books, 2002.
2. R.C. Martin, J.W. Newkirk, and B. Rao, "Taskmaster: An Architecture Pattern for GUI Applications," *C++ Report*, vol. 9, no. 3, 1997, pp. 12–14, 16–23; www.objectmentor.com/resources/articles/taskmast.pdf.
3. P.J.H. King, "Conversion of Decision Tables to Computer Programs by Rule Mask Techniques," *Comm. ACM*, vol. 9, no. 11, 1966, pp. 796–801.
4. D.L. Parnas, J. Madey, and M. Iglewski, "Precise Documentation of Well-Structured Programs," *IEEE Trans. Software Eng.*, vol. 20, no. 12, 1994, pp. 948–976.
5. D. Knuth, *The Art of Programming*, Addison-Wesley, 1973.

Dave Thomas is cofounder of Bedarra Research Labs (www.bedarra.com) and the OpenAugment Consortium (www.openaugment.org), and an adjunct professor at Carleton University and the University of Queensland, Australia. He's also a founding director of AgileAlliance.com and founder of Object-Technology International (www.oti.com). Contact him at dave@bedarra.com; www.davethomas.net.

CALL FOR ARTICLES:

Software Testing

Submission deadline: 1 November 2005

Publication: July/August 2006

The importance of testing in the software development process is widely accepted throughout the software community. However, software consumers and organizations sustain high losses due to defective software, which means that this is no straightforward process.

This issue will offer *IEEE Software* readers practical and proven solutions that help them to effectively and efficiently address their testing needs. We will focus on unit testing as one of the first and crucial aspects for V&V. Some unit testing-related issues that might be of interest to readers are:

- Automation of software testing
- Real experiences showing benefits of less widespread/applied testing techniques
- Empirical studies on testing techniques
- Trade-off analysis of testing techniques effectiveness and efficiency
- Improving testing relationships (management, users, teams, developers)
- Relevant testing metrics
- Best practices for testing in particular domains
- Landscape and trends for testing standards
- Experiences in testing process improvement

Manuscripts must not exceed 5,400 words, including figures and tables, which count for 200 words each. Submissions in excess of these limits may be rejected without refereeing. The articles we deem within the theme's scope will be peer-reviewed and are subject to editing for magazine style, clarity, organization, and space. We reserve the right to edit the title of all submissions. Be sure to include the name of the theme you are submitting for.

For detailed author guidelines and submission details, see www.computer.org/software/author.htm#Submission or email software@computer.org. To submit, go to <http://cs-ieee.manuscriptcentral.com/index.html>.

IEEE Software

Please contact the guest editors for more information about your idea's relevance to the topic area:

Natalia Juristo, natalia@fi.upm.es • Ana M. Moreno, ammoreno@fi.upm.es • Wolfgang B. Strigel, strigel@qalabs.com