

Black Box Software Testing Fall 2005

GUI REGRESSION AUTOMATION

by

Cem Kaner, J.D., Ph.D.

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

Copyright (c) Cem Kaner & James Bach, 2000-2004

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Overview

- The GUI regression test paradigm
- Engineering regression automation: Some successful architectures
- Planning for near-term ROI
- 30 common mistakes
- Questions to guide your analysis of requirements

Acknowledgment

Much of the material in this section was developed or polished during the meetings of the Los Altos Workshop on Software Testing (LAWST). See the paper, “Avoiding Shelfware” for lists of the attendees and a description of the LAWST projects.

The regression testing paradigm

This is the most commonly discussed automation approach:

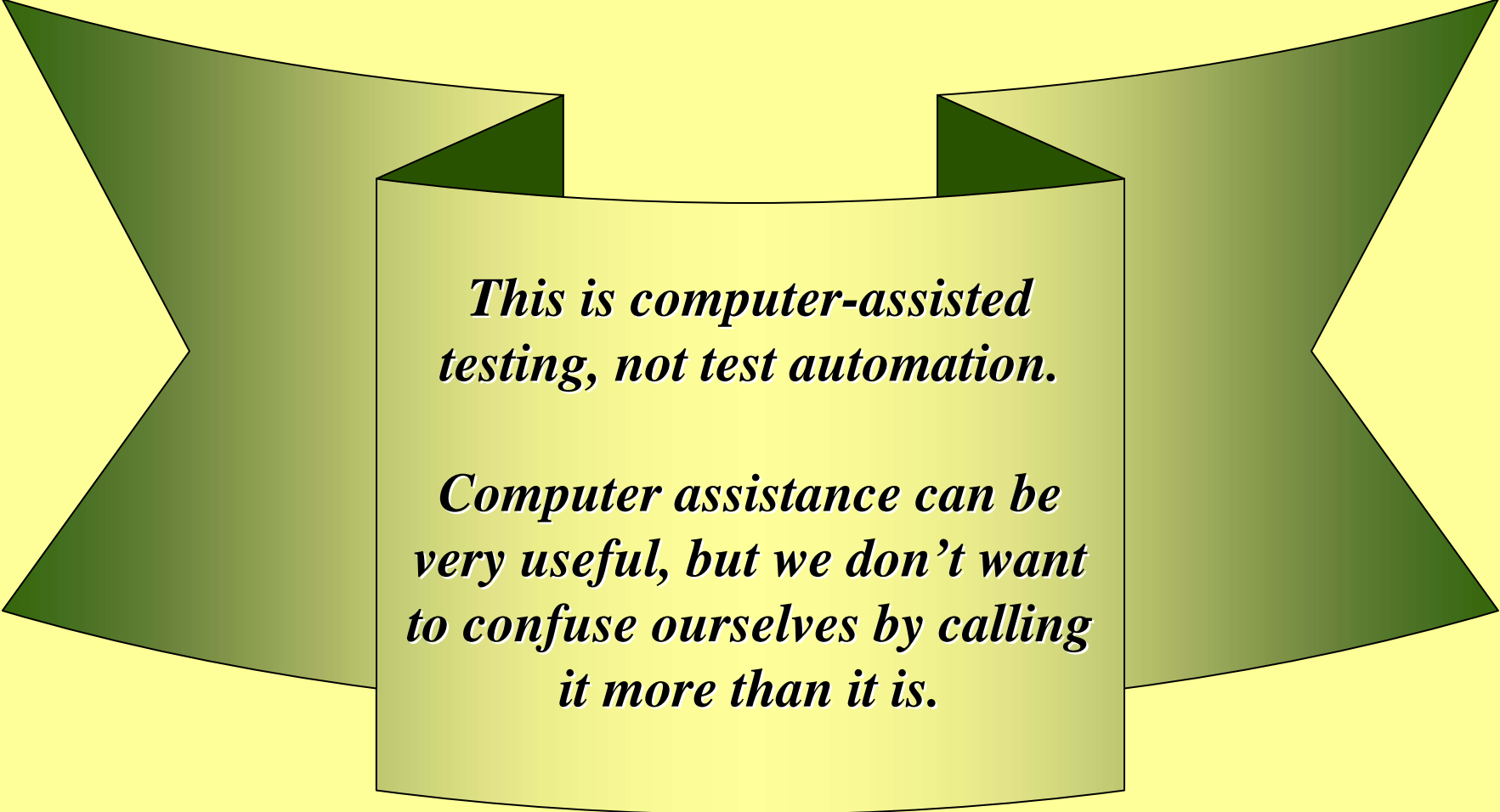
1. Create a test case
2. Run it and inspect the output
3. If the program fails, report a bug and try again later
4. If the program passes the test, save the resulting outputs
5. In future tests, run the program and compare the output to the saved results. Report an exception whenever the current output and the saved output don't match.

Costs and Benefits of Regression Automation

First, is this really automation?

- Analyze product -- Human
- Design test -- Human
- Run test 1st time -- Human
- Evaluate results -- Human
- Report 1st bug -- Human
- Save code -- Human
- Save result -- Human
- Document test -- Human
- Re-run the test -- **Machine**
- Evaluate result -- **Machine** plus human
if there's a mismatch
- Maintain result -- Human

**Woo-hoo! We really get the machine to do a *whole lot* of our work!
(Maybe ... but not this way.)**



*This is computer-assisted
testing, not test automation.*

*Computer assistance can be
very useful, but we don't want
to confuse ourselves by calling
it more than it is.*

Cost and benefit: The fundamental equations?

*Can we really
model the costs
and benefits in
terms of these
equations?*

Manual testing cost = Manual preparation cost + (N x Manual execution cost) ???

Automated testing cost = Automated preparation cost + (N x Automated execution cost) ???

Of course not. They treat ...

- Maintenance costs as non-existent
- The information gained from manual and automated tests as comparable
- The incremental benefits as constant. Is the Nth use *really* as valuable as the 2nd?

Direct costs

- **Test case creation is expensive.** Estimates for individual tests run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (custom controls).
- **Automated test creators get paid more** (on average) than comparably senior manual test creators.
- **Licensing costs can be quite high.** You may have to buy a license for any programmer who wants to *replicate* any bug exposed by these tests.
- **Maintenance costs can be enormous.** Brian Marick estimates that the cost of revising (including figuring out) a typical GUI test equals the cost of programming it fresh.
- **Development costs go beyond the test cases.** To bring maintenance costs under control, many groups will develop test harnesses.



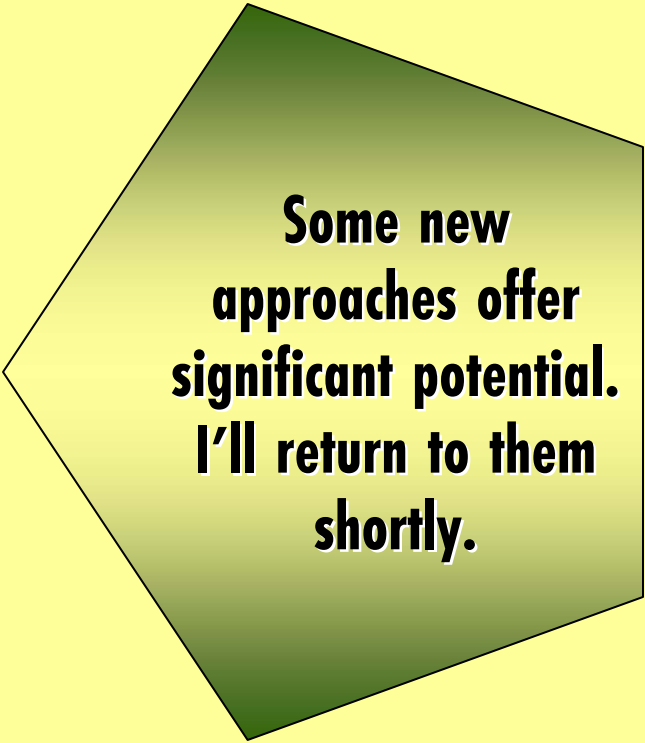
***You can
bring these
costs under
control, but
it will take
strategy,
investment
and work.***

Indirect costs: Consequences & opportunity costs

- It typically takes longer per test to create an automated GUI-level test than to create a manual test:
 - You have to design, develop, test, document and store the test, not just think it up and do it.
 - Consequence: You probably won't develop as many tests per week. So ...
 - You'll find some bugs later than you would have with manual testing (because you'll develop the tests later). It often costs more to fix bugs later.
 - You probably have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation, so they aren't doing bug hunting or training other testers.

Effectiveness?


- Relying on **pre-designed** tests carries the 20 questions problem.
 - Defining a test suite before you know a program's weaknesses is like playing 20 questions where you have to ask all the questions before you get your first answer.
- A common estimate at LAWST (and elsewhere) has been that the GUI regression tests found about 15% of the total bugs found in testing.
 - The numbers vary widely, but they are disturbing. See our first discussion of regression testing.



**Some new
approaches offer
significant potential.
I'll return to them
shortly.**

Benefits of regression come later

- A good set of regression tests might eventually provide significant value
 - Smoke tests, change detection in low-risk areas, patches, localization ...
- But because they take so long to develop, *those* benefits are delayed.
- Consider the version (not build) when these tests are created:
 - Rerun tests *in this version* probably have lower power than new tests
- Several senior practitioners have estimated a 3-version (e.g. 3-year) time to recover the investment. They posit low reuse value in the first version and include maintenance costs.



**Maintainability is
essential for
recovering an
investment in
regression tests.**

Test automation requires software engineering

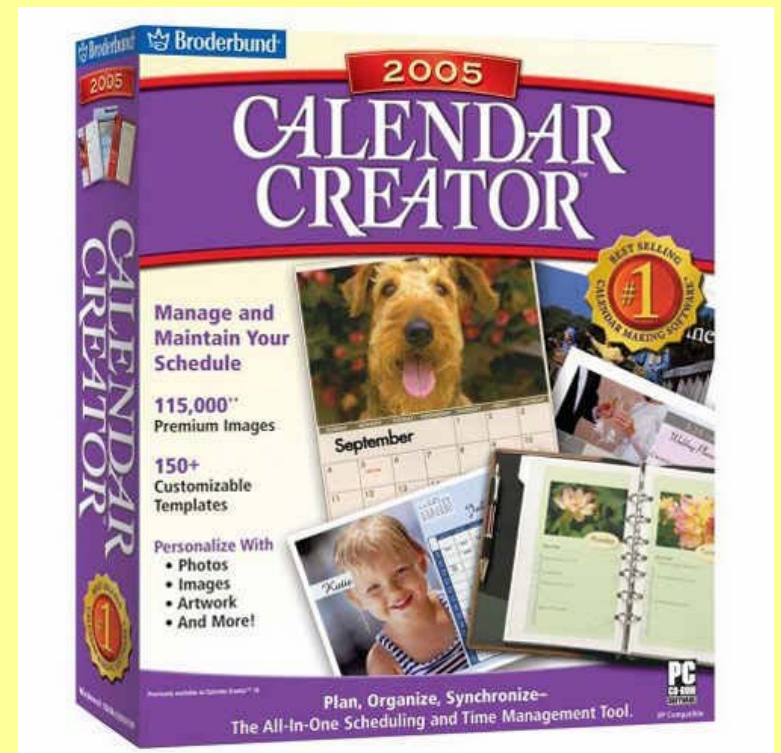
- Win NT 4 had 6 million lines of code, and 12 million lines of test code
- Common (and often vendor-recommended) design and programming practices for automated testing are appalling:
 - **Embedded constants**
 - No modularity
 - **NO SOURCE CONTROL**
 - No documentation
 - *No requirements analysis*



Engineering Regression Automation

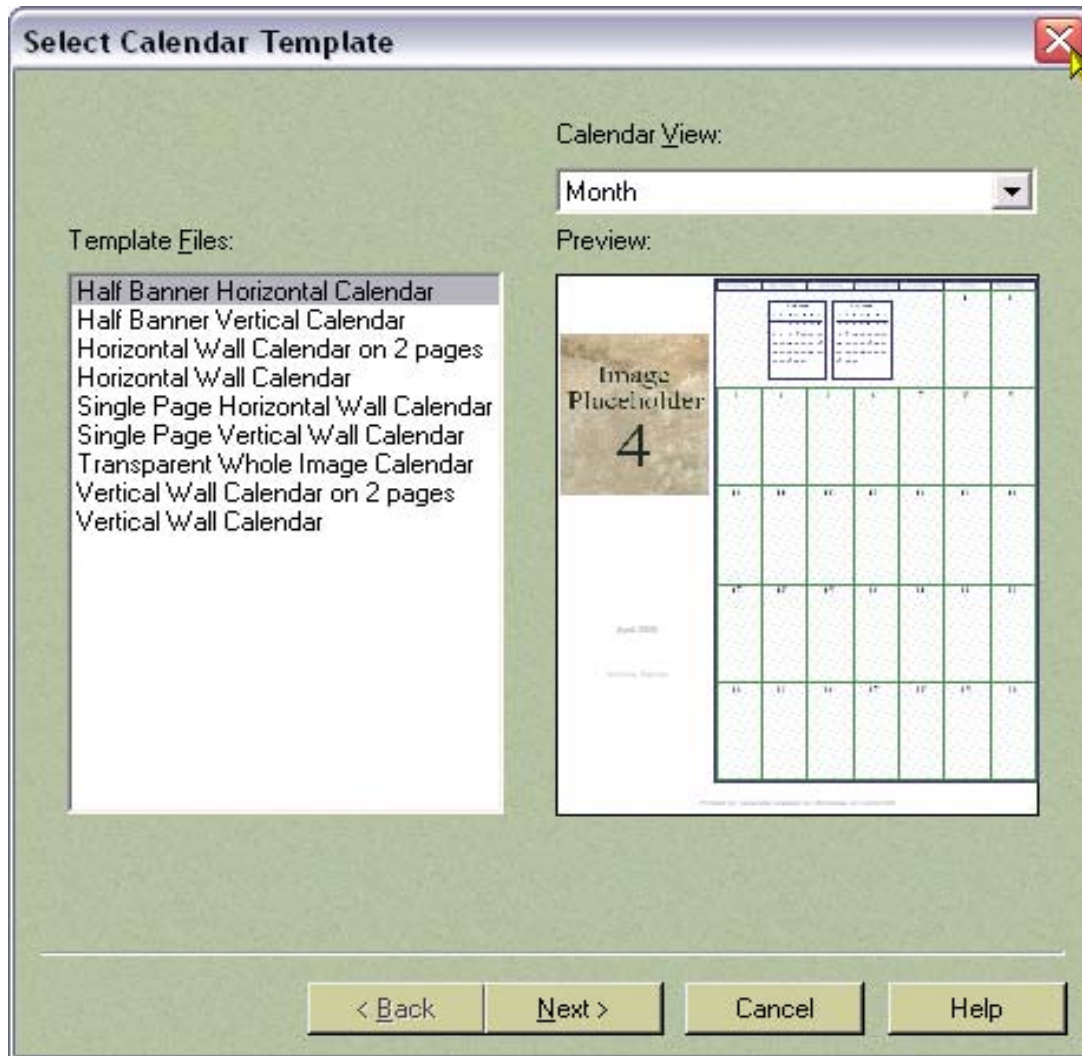
A simple(?) example

- Let's look at Calendar Creator
 - Consumer market
 - All you're really trying to do is capture events and lay them out on a pretty calendar.
 - How complex could that get?

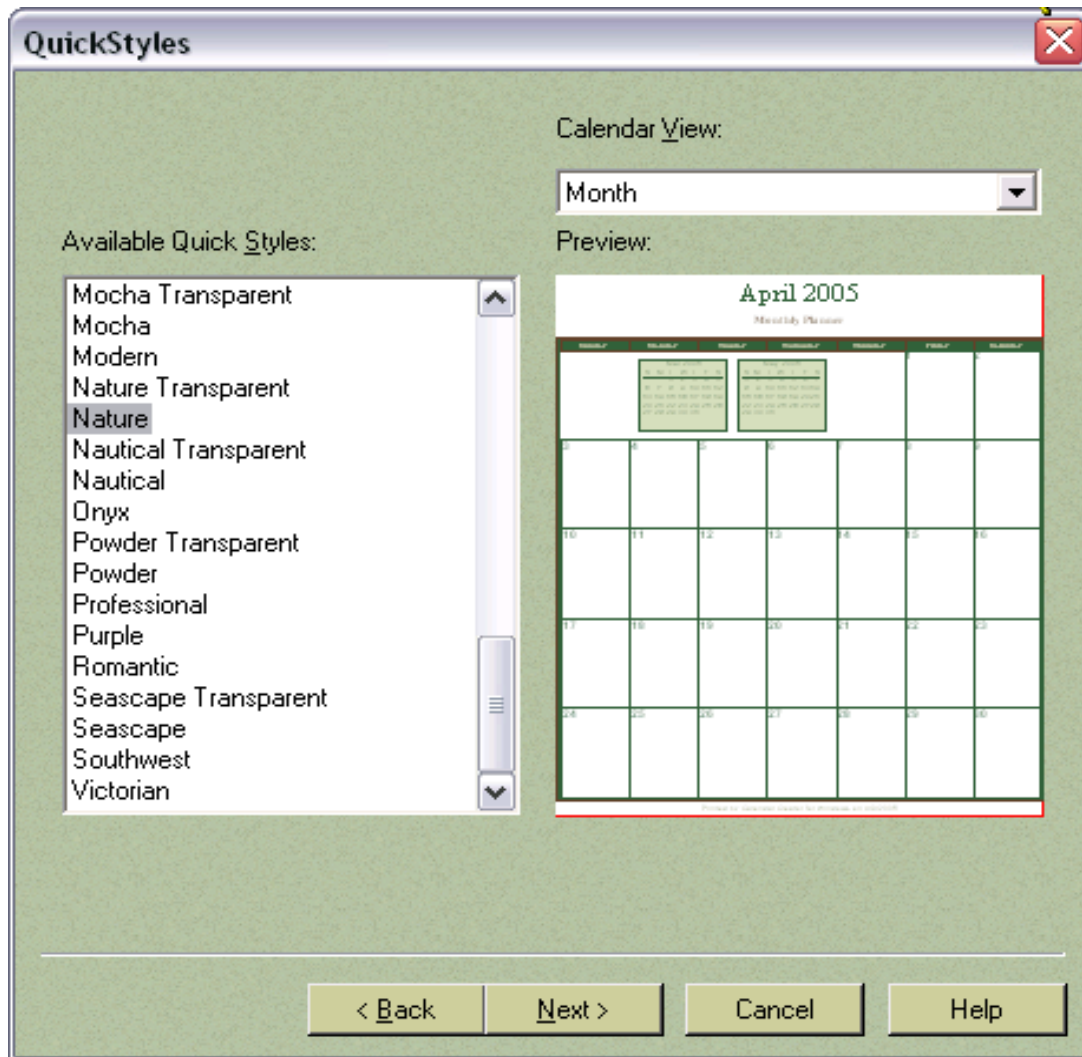


Calendar Creator

- Here are just **some** of the things we can vary in these calendars
 - **Different languages for headings**
 - **Lots of basic layouts, or you can grow your own**



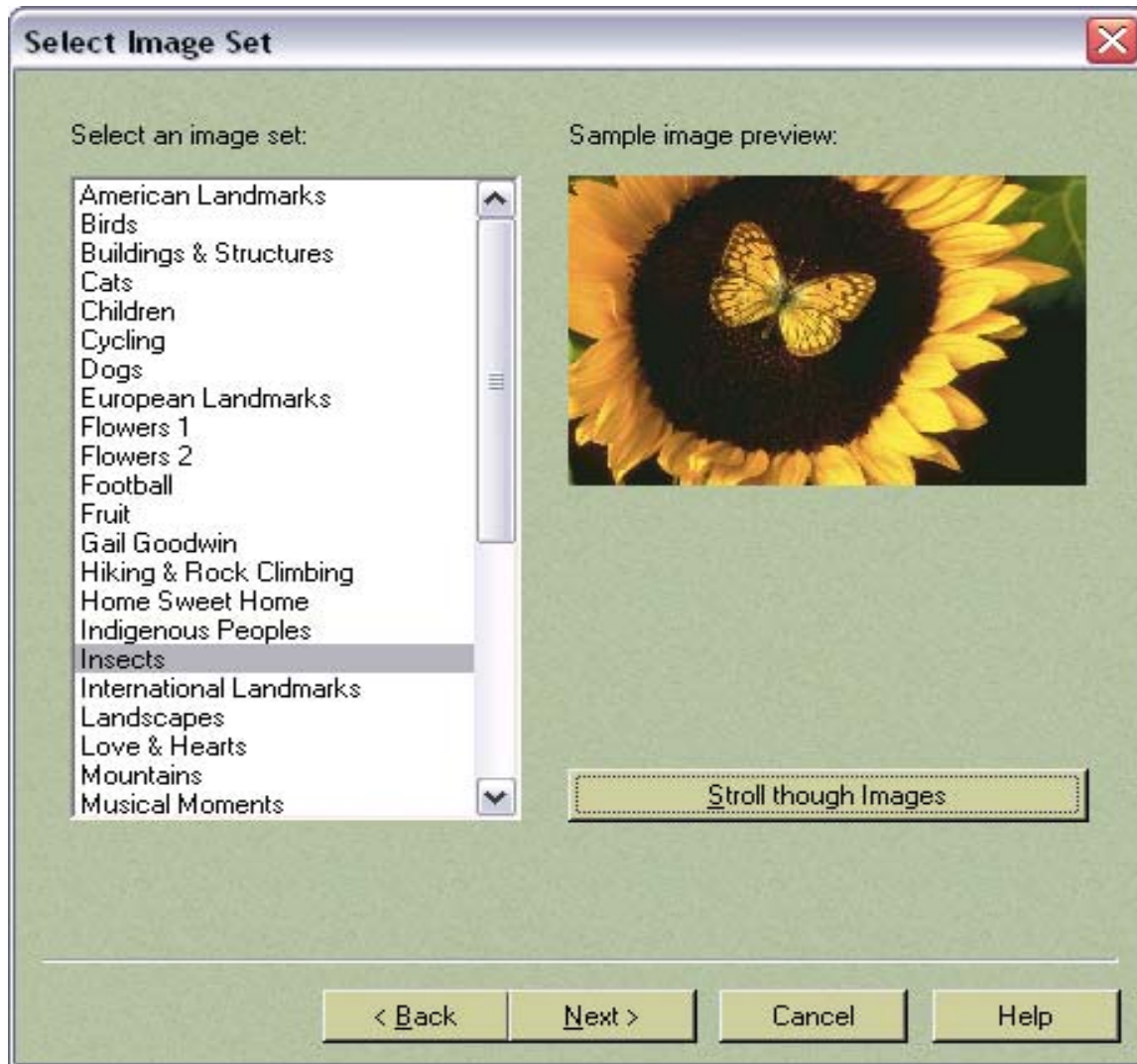
- Lots of basic layouts – or you can grow your own
- “**Templates** are pre-formatted layouts that include artwork and text styles. You can create new calendars from templates or apply a template to an existing calendar.” (CC help)



“QuickStyles are pre-formatted font and color schemes that can be applied to any calendar, even one that was created using a template. The QuickStyle you select simply replaces all existing font and color options in the active calendar. Once you’ve applied a QuickStyle, you can further customize the objects on your calendar to create dramatic effects. You can even create and save your own QuickStyles.” (CC Help)

Calendar Creator

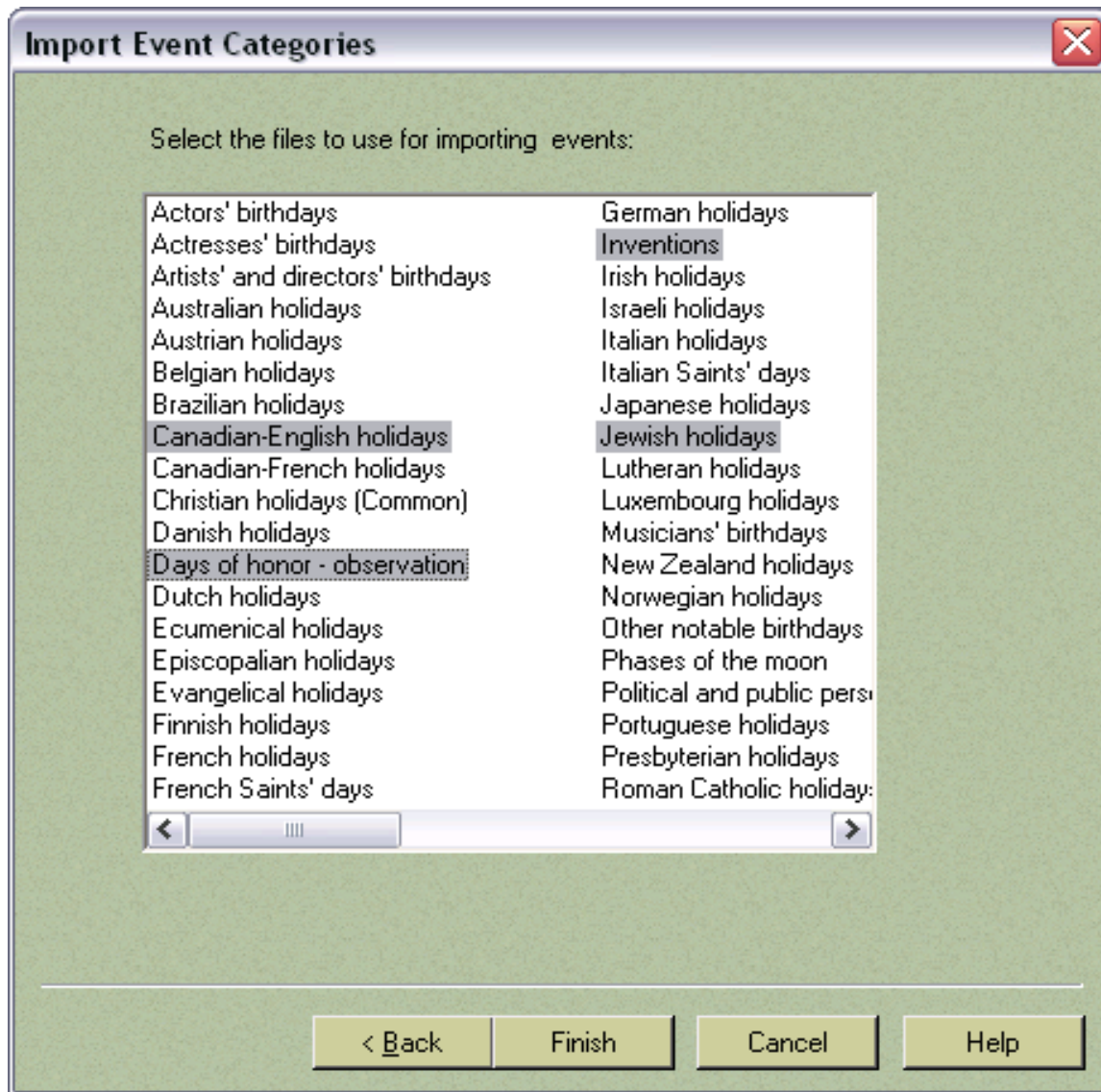
- Here are just **some** of the things we can vary in these calendars
 - Different languages for headings
 - Lots of basic layouts, or you can grow your own
 - **Different color schemes**
 - **Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)**



- Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)
- (I'm not showing the library / file navigation tabs, but they exist)
- One or more pictures for the month, on the top or sides

Calendar Creator

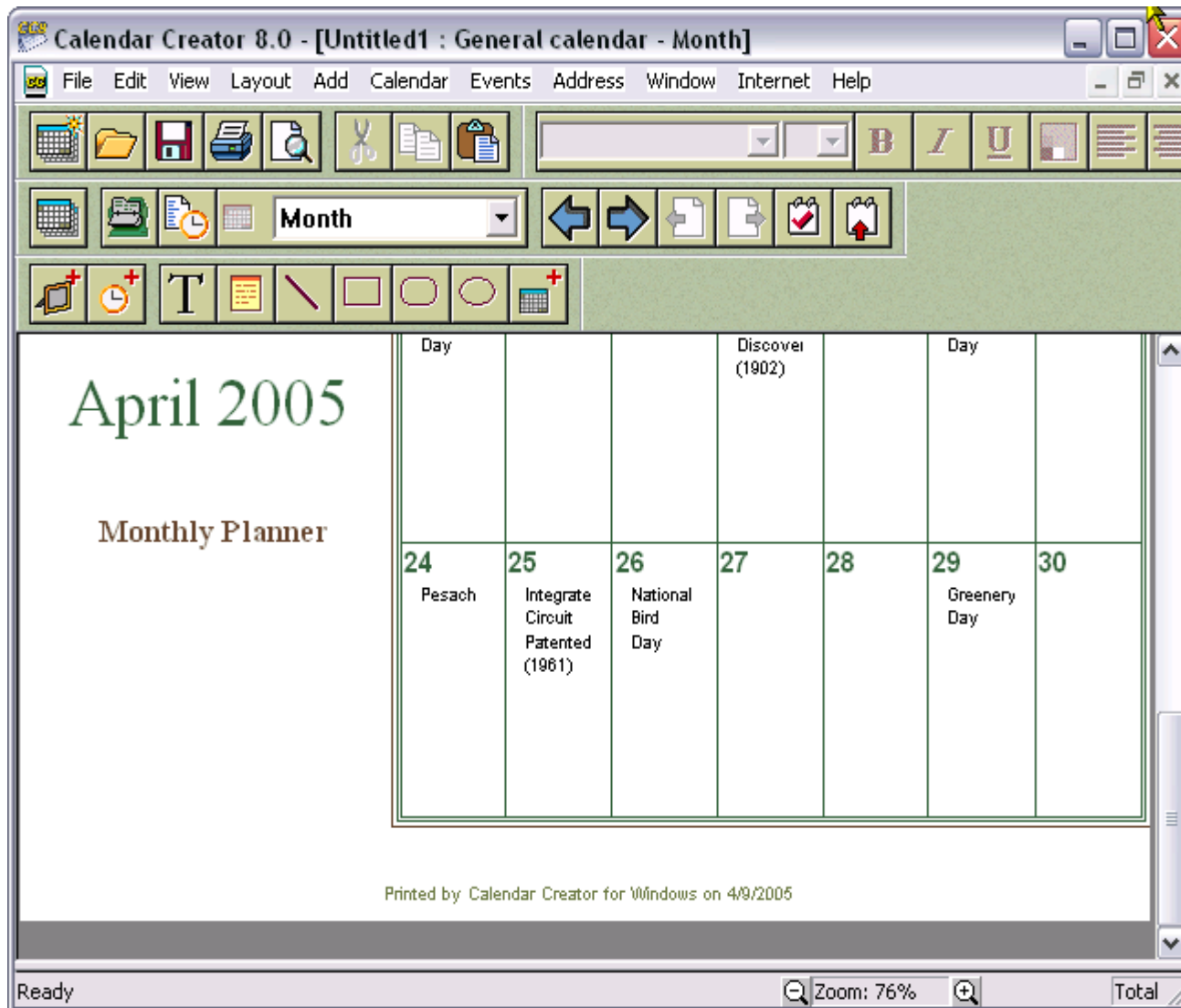
- Here are just **some** of the things we can vary in these calendars
 - Different languages for headings
 - Lots of basic layouts, or you can grow your own
 - One or more pictures for the month, on the top or sides
 - Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)
 - **One or more pictures for the month, on the top or sides**
 - **5 or 7 days per week, weeks start any day**
 - **Add lots and lots of events to those days**



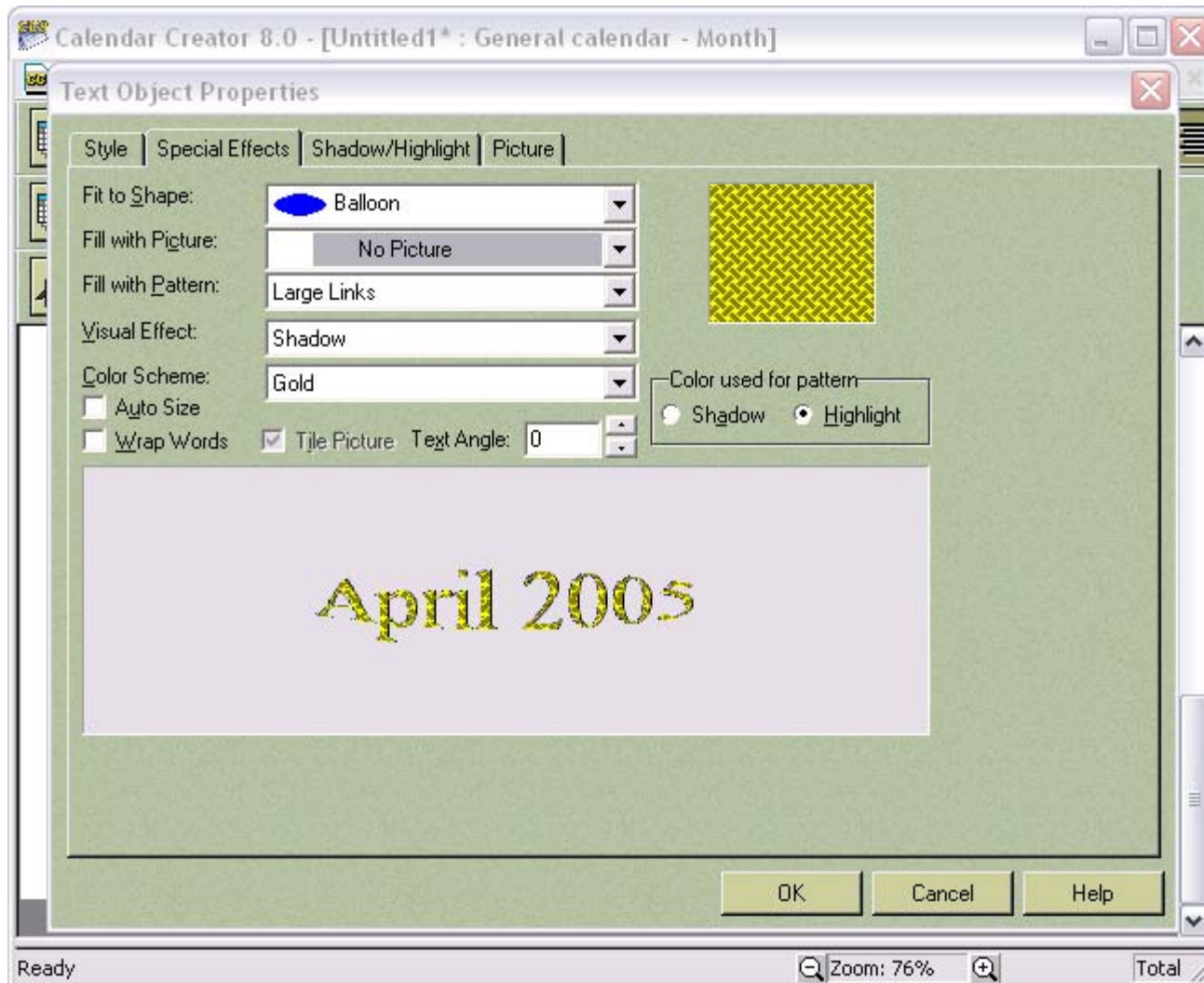
We imported US holidays on the video. Here we're adding dates for famous inventions, Canadian-English holidays, days of honor, and Jewish holidays. You can add your own events too, and make your own collection of them.

Calendar Creator

- Here are just **some** of the things we can vary in these calendars
 - Different languages (French, German, Russian, etc.) for headings
 - Lots of basic layouts, or you can grow your own
 - Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)
 - One or more pictures for the month, on the top or sides
 - 5 or 7 days per week, weeks start any day
 - Add lots and lots of events to those days
 - **Reformat and reposition the headings**



Reformat and reposition headings—from here...



Reformat it. I added a shadow to the text, filled the text with a “large links” pattern, changed its color to gold, and then fit the text to a balloon shape.



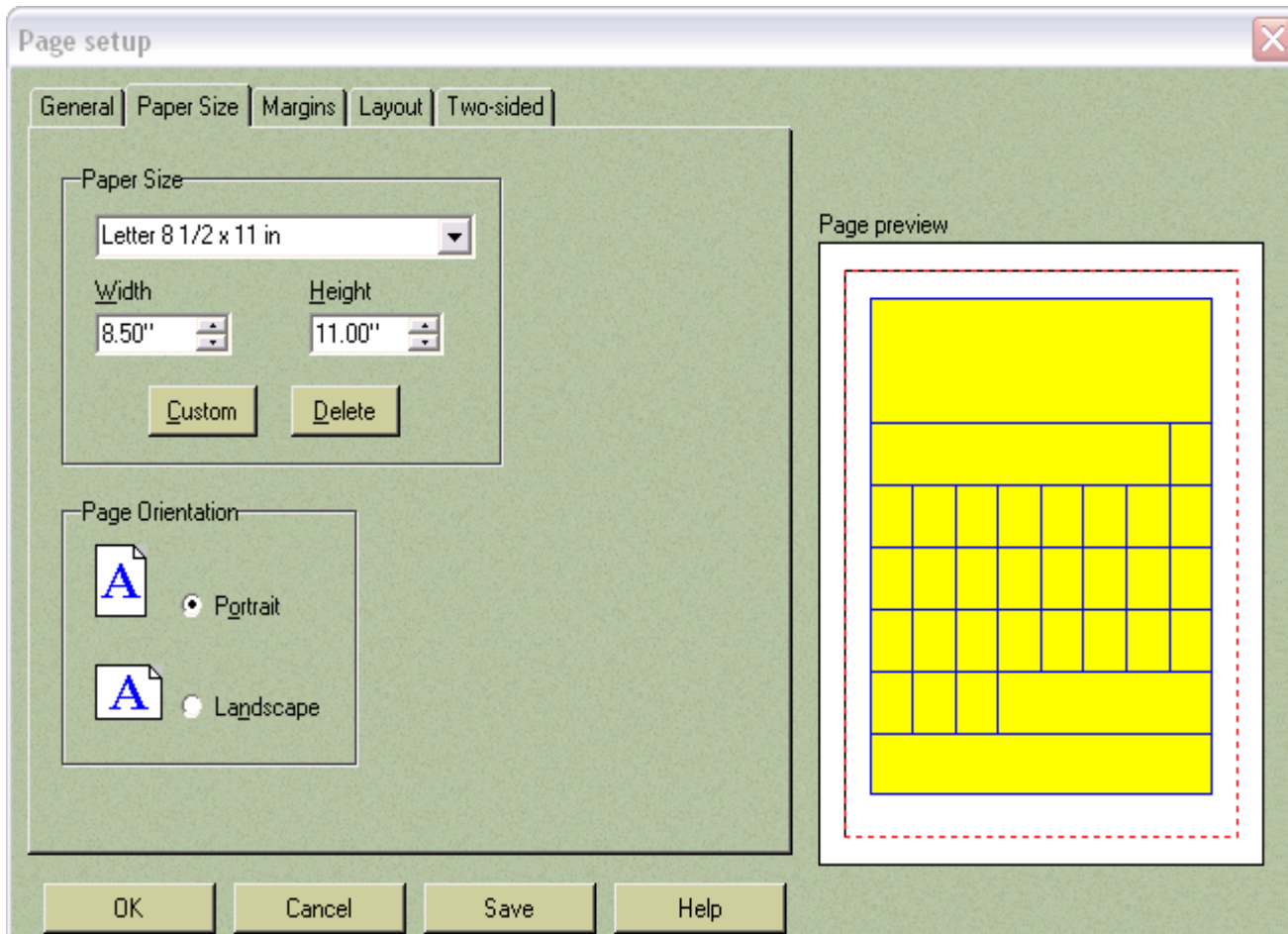
- Here's the calendar page for April, so far
- Notice there are 2 events—matches invented **and** World Health Day on April 7.
- We can have more. They overflow to the bottom of the page if they don't fit. (Or to the **next** page if the bottom fills up.)



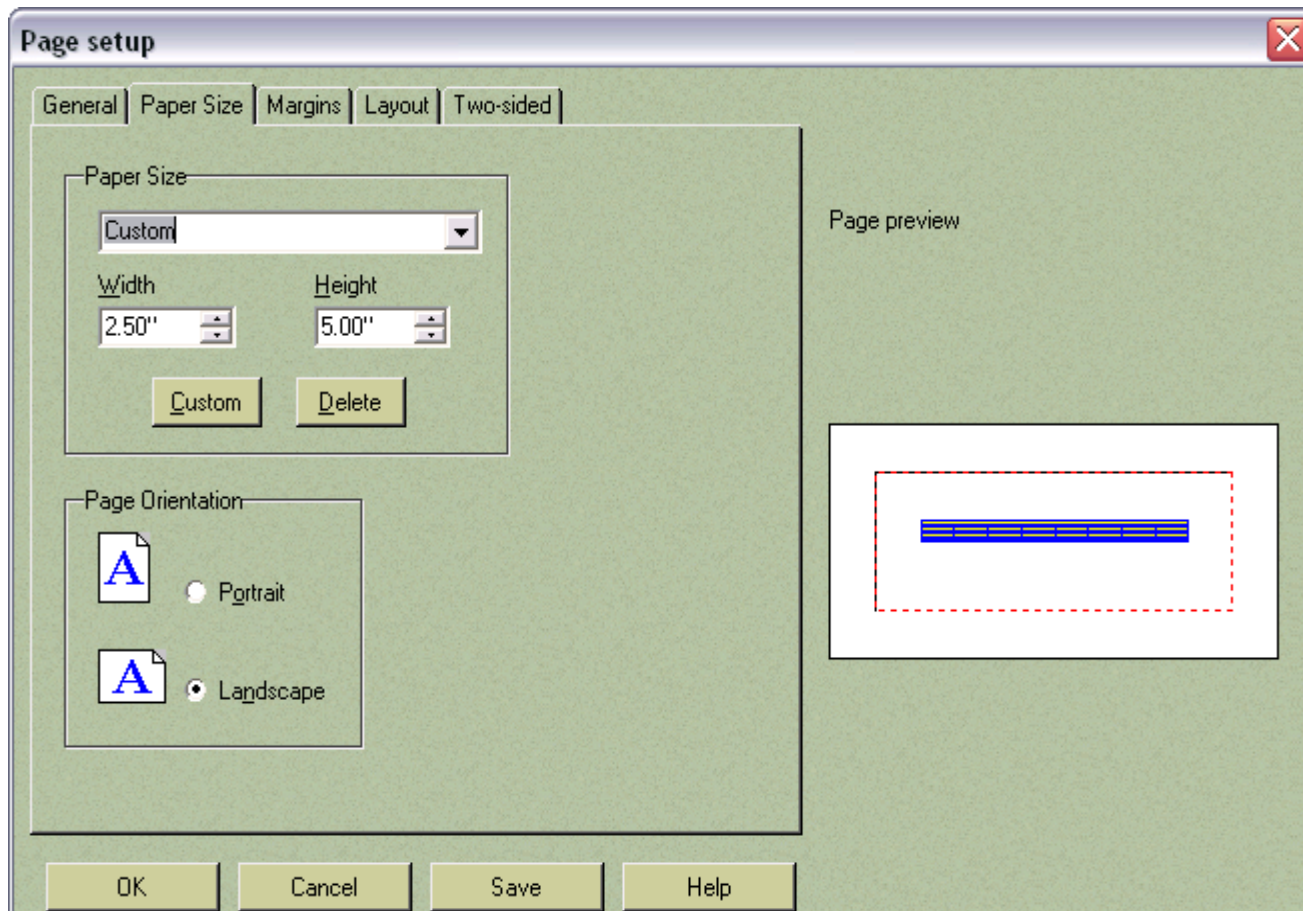
I moved April up and “Monthly Planner” down (monthly planner is still selected).

Calendar Creator

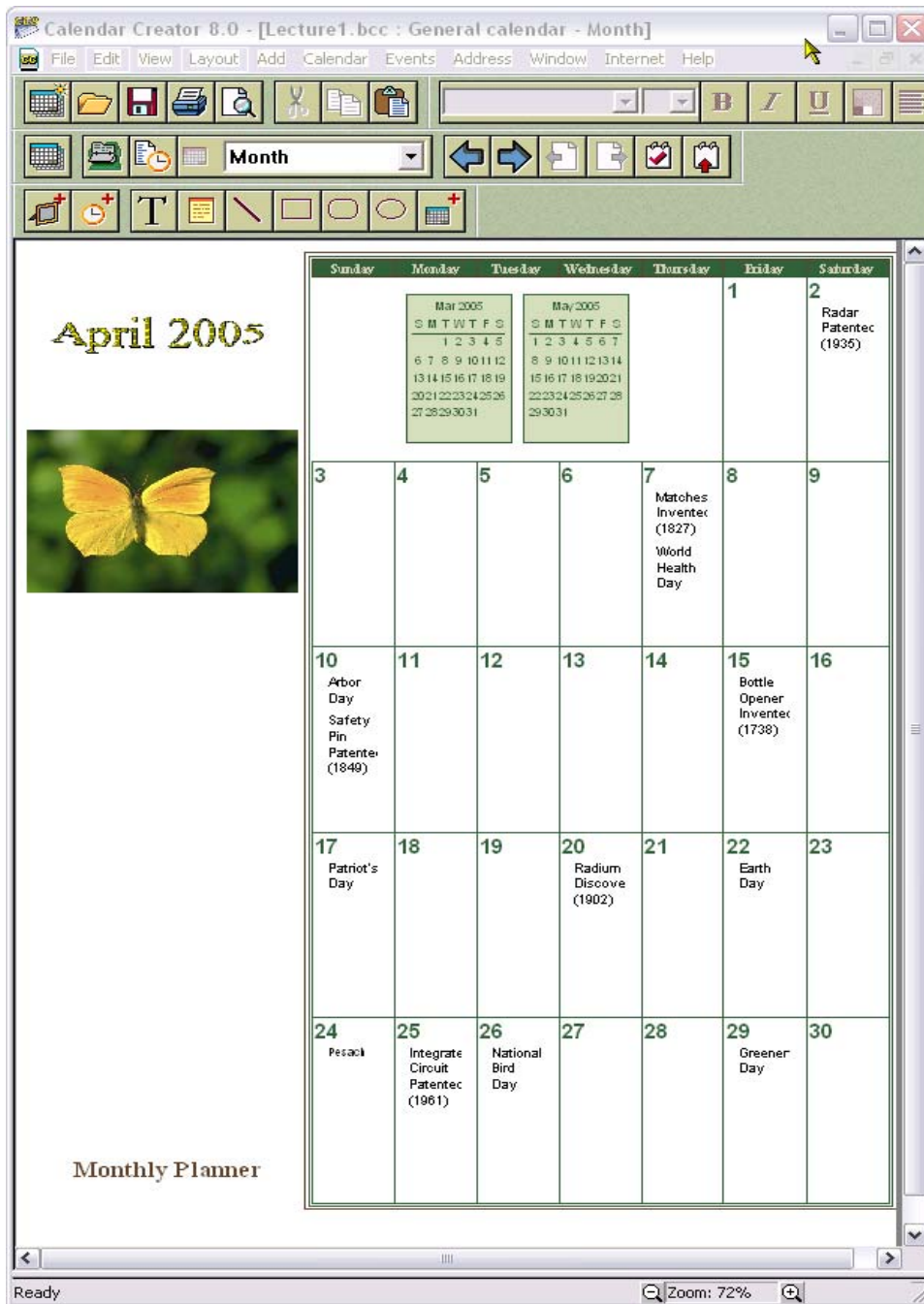
- Here are just **some** of the things we can vary in these calendars
 - Different languages (French, German, Russian, etc.) for text
 - Lots of basic layouts, or you can grow your own
 - Reformat and reposition the headings
 - One or more pictures for the month, on the top or sides
 - Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)
 - 5 or 7 days per week, weeks start any day
 - Add lots and lots of events to those days. One or more events on any day. Different typefaces and sizes for each event
 - Reformat and reposition the headings
 - **Zero, one, or more pictures on any day**
 - **Width and height of the day (in the calendar table) depend on size of the paper and the size of graphics (if any) beside or above or below the table**



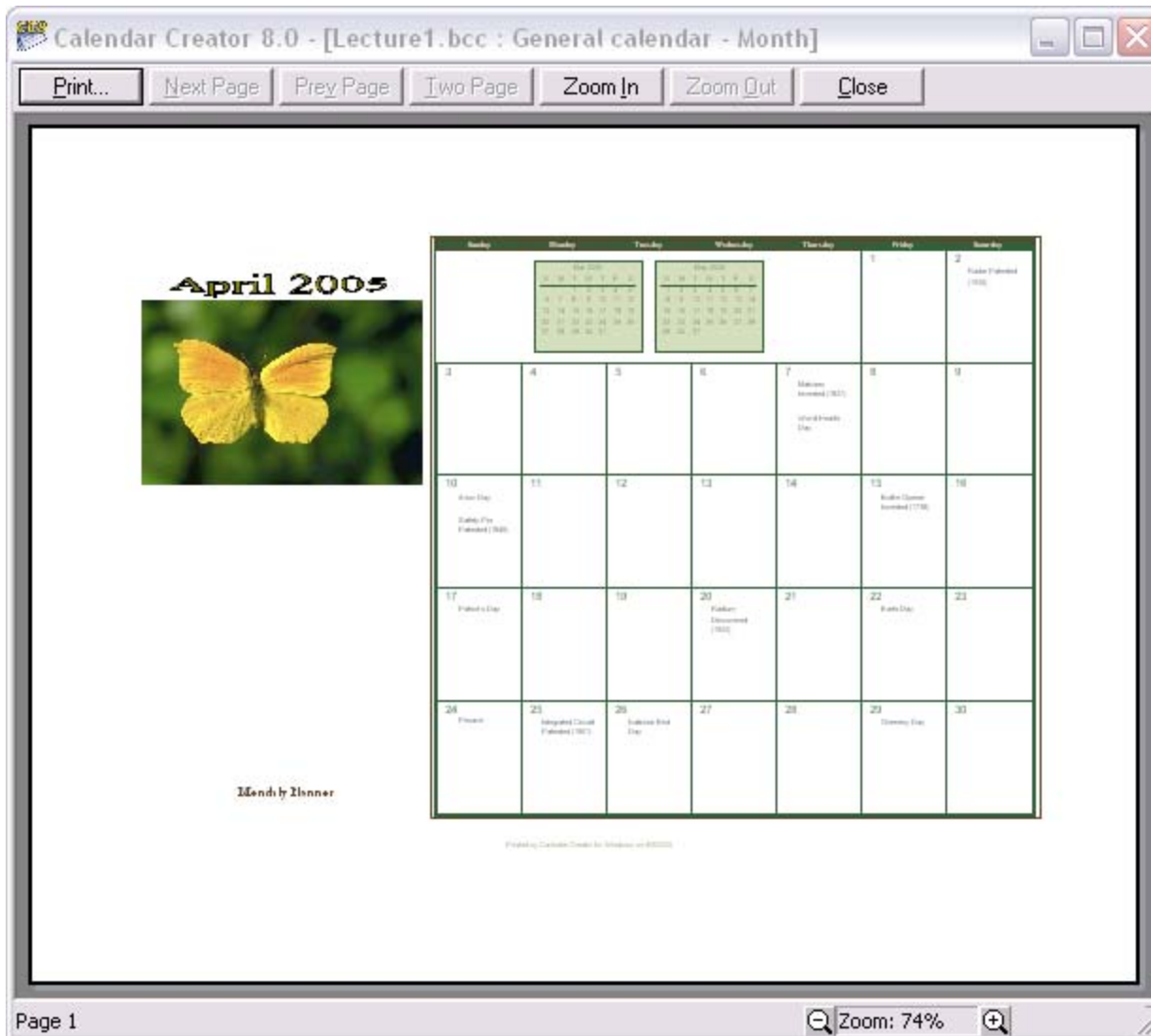
- Let's make the page smaller.
- We've got text at the top and the bottom...



Changed from 8.5 x 11 to 2.5 x 5.0 and from portrait to landscape



Page setup had no effect on the calendar onscreen.



The print preview adjusts things reasonably

Calendar Creator 8.0

Your custom paper stock may not
be supported by your printer.
Calendar Creator will try to print
on this paper stock anyway.
In the event that this fails, the default
paper stock will be used.

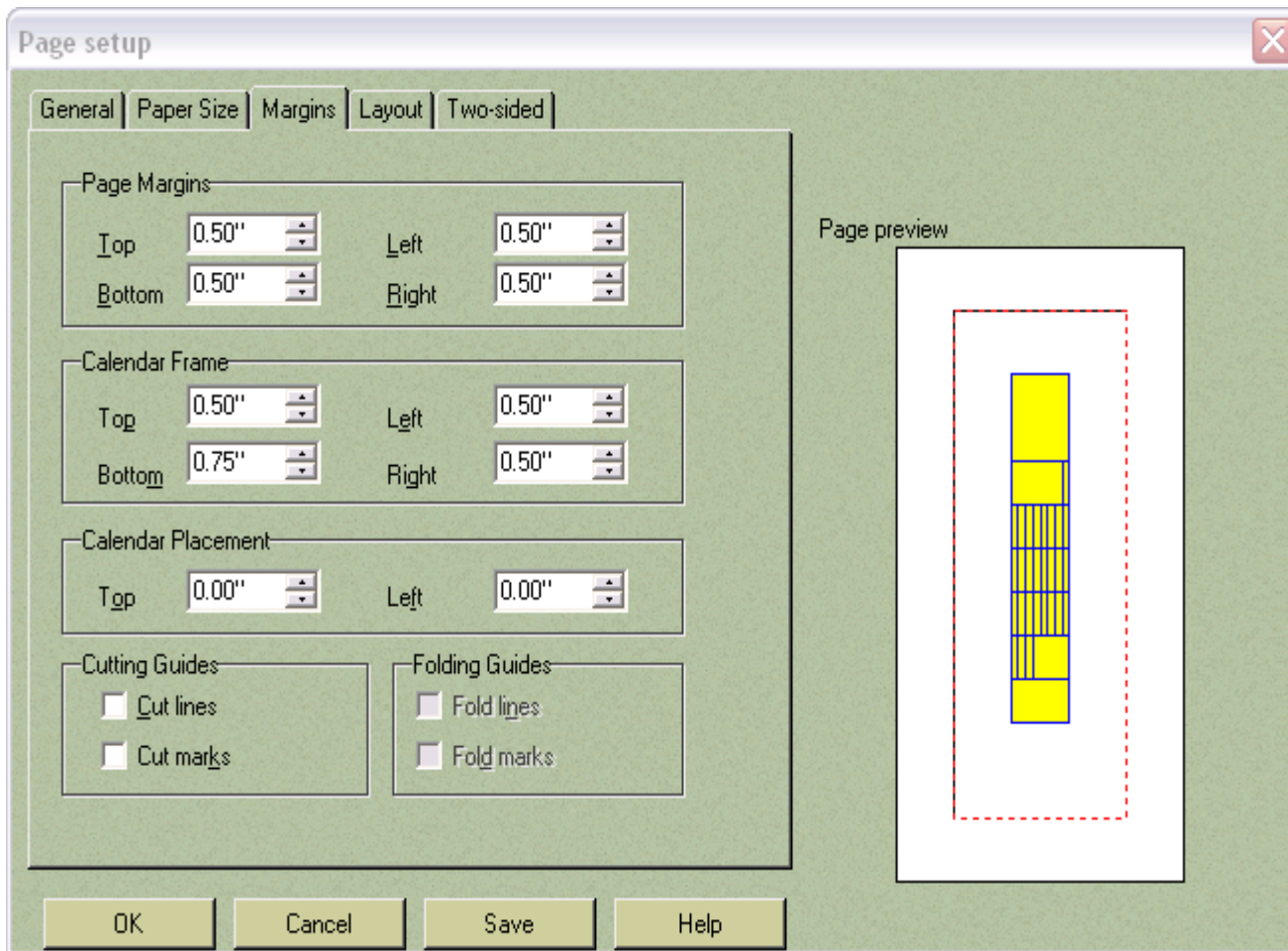
OK

- Print Preview gives this message.
- Looks like we'll get different messages depending on the printer.

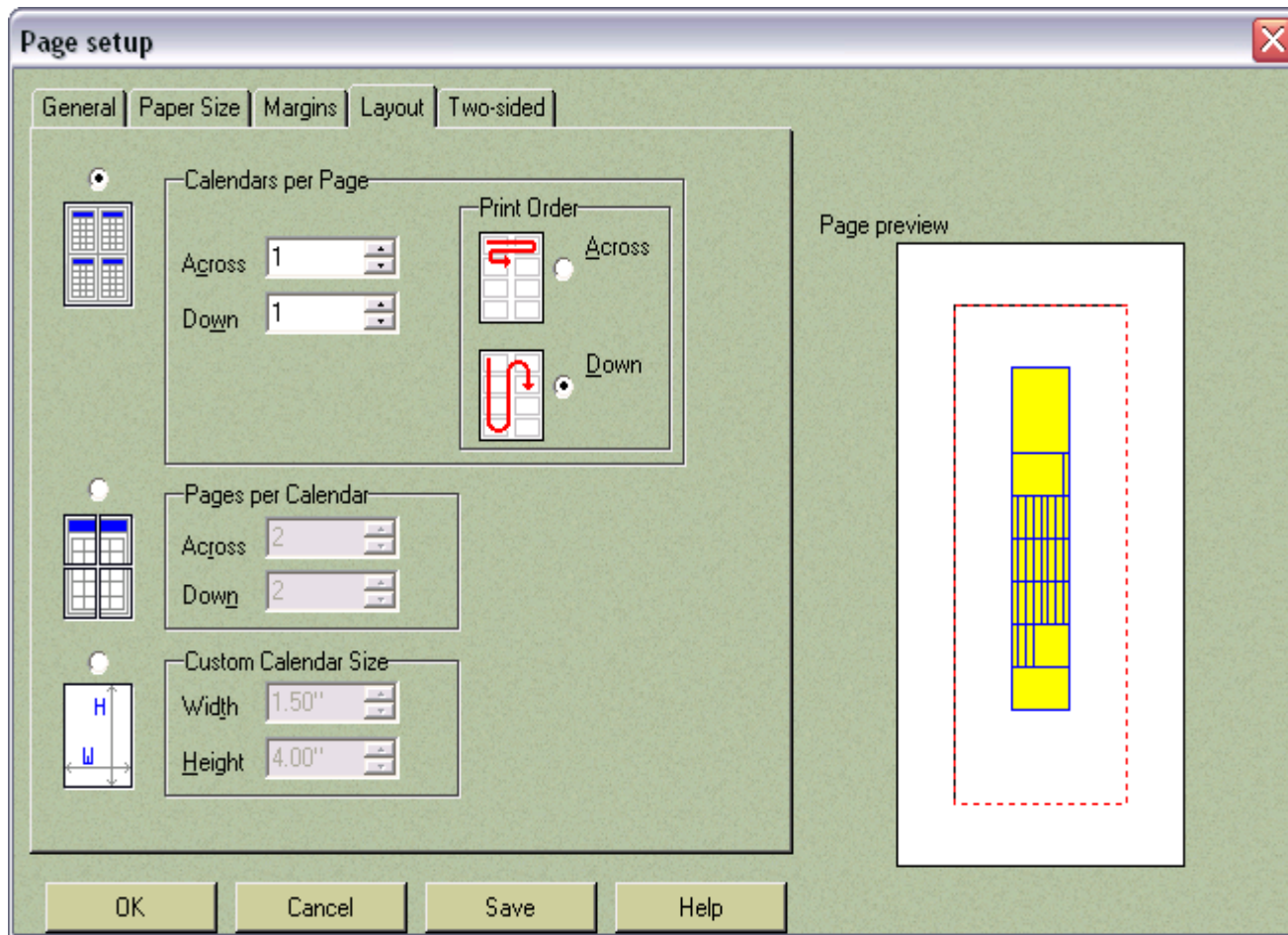
Calendar Creator

Here are just **some** of the things we can vary in these calendars

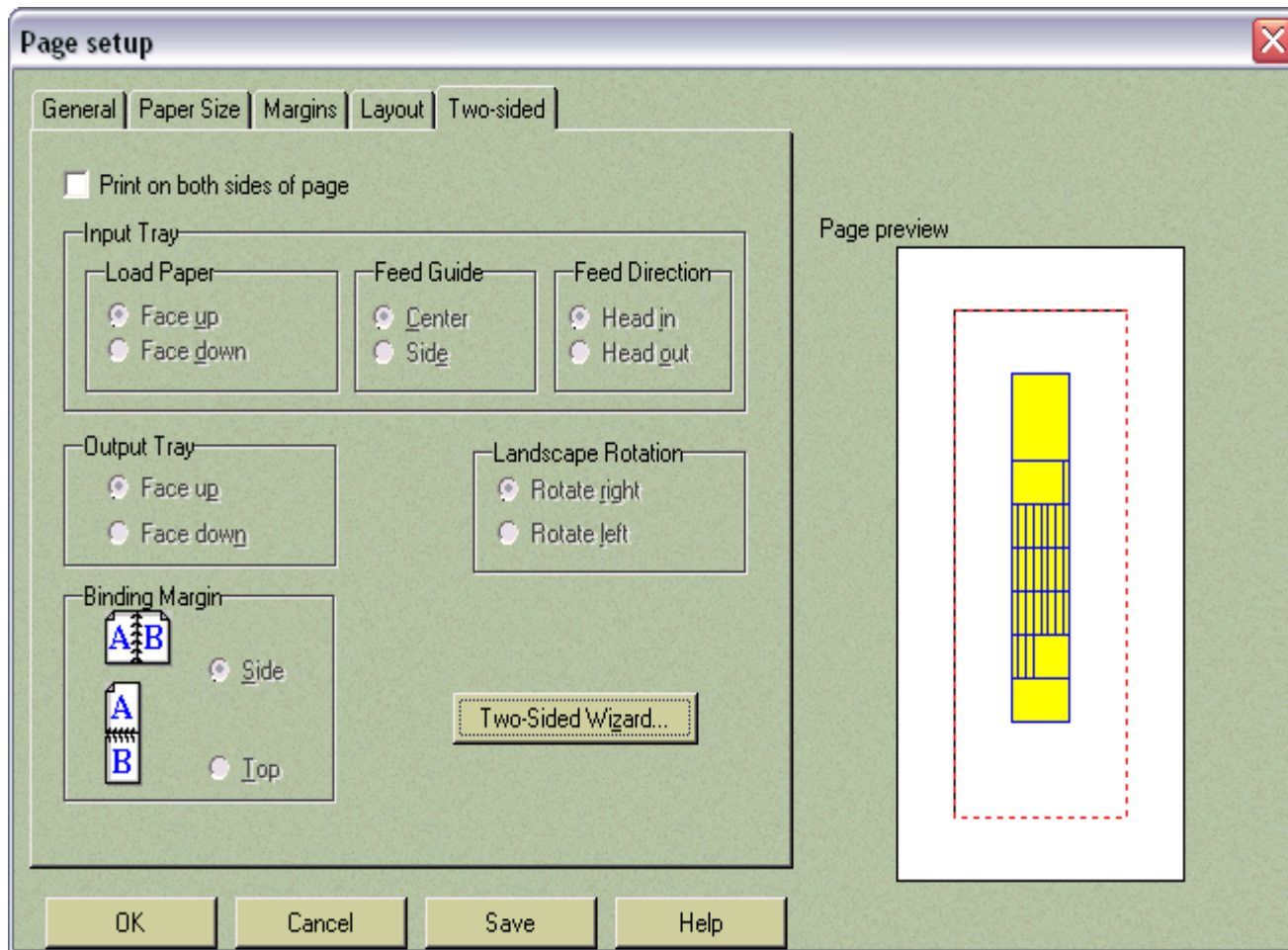
- Different languages (French, German, Russian, etc.) for text
- Lots of basic layouts, or you can grow your own
- Reformat and reposition the headings
- One or more pictures for the month, on the top or sides
- Lots of pictures (use theirs or your own clipart) (of various file types) (in various directories, maybe on a remote machine)
- 5 or 7 days per week, weeks start any day
- Add lots and lots of events to those days. One or more events on any day. Different typefaces and sizes for each event
- Reformat and reposition the headings
- Zero, one, or more pictures on any day
- Width and height of the day (in the calendar table) depend on size of the paper and the size of graphics (if any) beside or above or below the table
- **Change margins, spread a calendar across several pages, or shrink to fit several on one page when we print.**



Change several different margins



Spread a calendar across several pages, or shrink calendars to fit several on one page when we print.



If we're printing over several pages, we may want to print double-sided.

Calendar Creator

Here are just **some** of the things we can vary in these calendars

- Different languages (French, German, Russian, etc.) and text
- Lots of basic layouts, or you can grow your own
- Reformat and reposition the headings
- One or more pictures for the month and for the sides
- Lots of pictures (use theirs or your own) (of various file types) (in various directories, maybe on a web page or machine)
- 5 or 7 days per week, week starting on any day
- Add lots and lots of events for any days. One or more events on any day. Different types of events for each event
- Reformat and reposition the headings
- Zero, one or more pictures on any day
- Width and height of one day (in the calendar table) depend on size of the paper and the amount of graphics (if any) beside or above or below the table
- Change margins, spread a calendar across several pages, or shrink to fit several on one page when we print.

ENOUGH ALREADY! THE PAGE IS FULL!

Designing reusable tests

- We've just looked at three types of variation:
 - **How the program interacts with input devices** (picture files, event files—and a whole address database we haven't looked at)
 - **How the program interacts with output devices** (printers, but there are disks and displays too)
 - **How the world designs calendars** (and how you can use the program to make calendars that people in the world would consider functional and/or pretty)
- So let's start with tests associated with these.

Our goal is to partition the solution so changes to the software or environment require only narrow, focused changes to the tests.

Variation associated with input devices

- Build
 - Graphics libraries
 - Event datafiles
 - Address databases
- that live in
 - Different directories
 - On different drives
 - Local and remote
- and are in different states of repair
- and plan to make them available, unavailable, or to interrupt their availability during use

The program will have to deal with new kinds of inputs from new devices no matter how it is designed or implemented. What other input devices should we plan for?

Variation associated with output devices

- We don't need to know how **this** program will print to design tests of calendars that are bannered across pages (multiple pages that have to be pasted together).
- What other test ideas can we develop that depend on the characteristics of the printer or the paper or the program's interface to them?
 - Can we build a list of test ideas for variation among (e.g.) printers?
 - Is it possible to take an output and transform it before printing it, so that it is more challenging for the target printer?

We are planning for coping with changes in the capabilities of the output device, not (in these ideas) for changes in the program.

Variation in desired results

- How the world designs calendars
 - There are plenty of calendar designs in the world.
 - Some attributes vary from country to country (e.g. weekdays vertical or horizontal).
 - Designs involve taste and function—layout for a small DayTimer page will be very different from layout for a wall calendar.

We could build a large database of calendar designs, to use as tests, without once referring to any implementation details of the program.

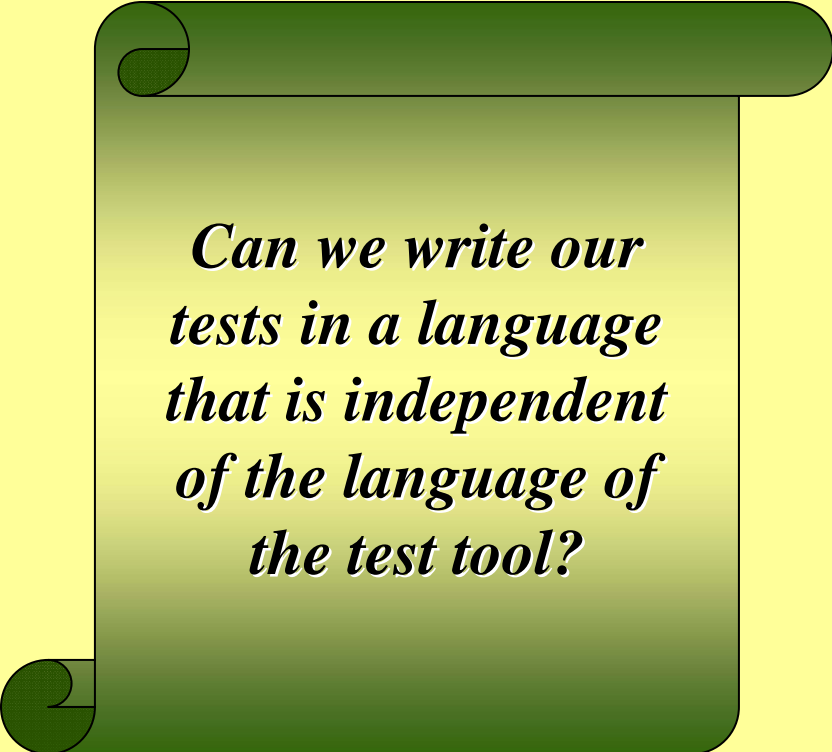
Another source of variation: The UI

- The presentation of a program can change without any change to its underlying capabilities. For example:
 - The name of any menu item or command or dialog can change.
 - Any item can move from one menu or dialog to any other menu or dialog.
 - The sequence of sub-tasks can change order, and some might be added or separated.
 - The effect of one choice on another can change.
 - The presentation of choices might depend on display bandwidth.
 - The presentation of choices might depend on the user interface (system-selected) language

Rather than trying to force people to lock down the UI so we can treat it like a set of constants, let's recognize that it is a collection of variables.

A fifth locus of variation: The test tool

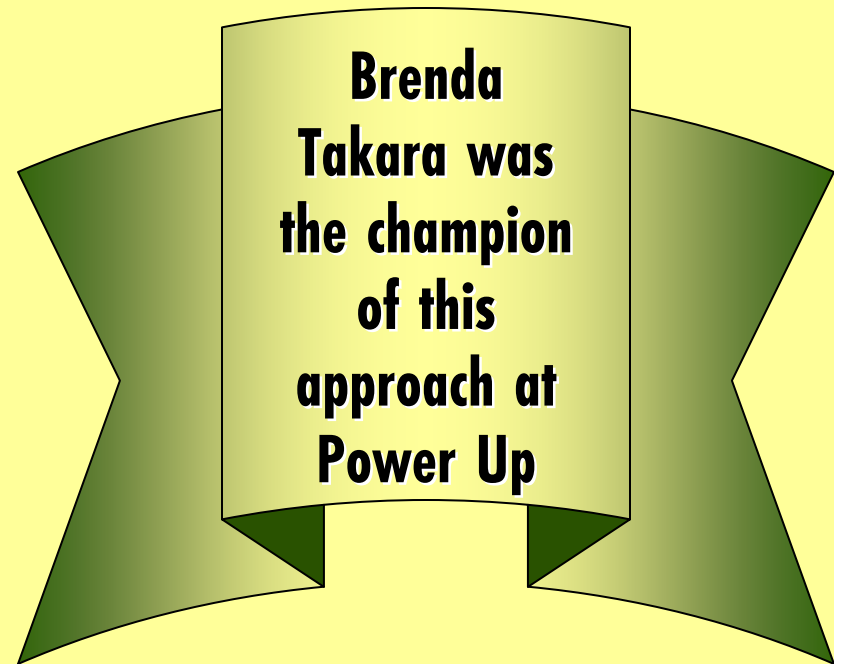
- What is the syntax of your test tool's programming language?
 - When did it last change?
- Have you ever tried to port your tests from one vendor's tool to another's?
 - Do you really want all your tests to be locked to a specific vendor because of the cost of porting to a new tool language?



Can we write our tests in a language that is independent of the language of the test tool?

A data-driven approach - 1

- Define your tests in a combination chart:
 - Attributes of the calendar go across the top of a very wide table
 - One row per test
 - The cell defines the value, such as
 - What month
 - What typeface / size / style
 - What event lists are loaded
 - Where the main calendar picture for that month is positioned
 - What the picture is



A data-driven approach - 2

- The combination chart is parsed and executed by an interpreter
 - Select a row (e.g. Row 3). This is a specification of a test.
 - The interpreter reads the values of the cells in Test 3, one column at a time.
 - A distinct method (test script) is associated with each column
 - Example: If the column specifies a date, the method navigates the calendar to that date.
- **The primary chart includes values specific to calendars, not to the devices, UI, or test tool.**
- You might extend the chart (or use a linked chart) to specify device-dependent tests.

If the program under test changes, change the interpreter's methods.

The table's tests can stay the same, no matter how many changes you make to the user interface.

A data-driven approach - 3

- It often makes sense to write a wrapper around every one of the test tool's commands.
- So,
 - rather than directly calling the tool's MoveObjectUp command
 - you might create your own MyMoveObjectUp command
 - ⇒ *that calls* MoveObjectUp
 - and then write all of *your* code to call MyMoveObjectUp.

**If vendor syntax changes,
or if you change vendors,
rewrite MyMoveObjectUp
to call whatever it has to
call to preserve its old
functionality.**

**All your methods that
called MyMoveObjectUp
can stay the same.**

A data-driven approach - 4

- An higher-level execution engine runs the test process:
 - Reads in the appropriate table(s)
 - Points the interpreter at the table(s)
 - Sets up output
 - In our case, for each test:
 - Print Section 1 (probably 1 page)
 - » Name the test (title of calendar)
 - » list all the settings (variables and their values)
 - Print Section 2 (probably 1 page)
 - » The test result (the calendar)

Note: I have simplified and extended this from the original Power Up work, to make a clearer teaching example.

A data-driven approach - 5

- In principle, you could extend this:
 - Include a “Results Saved” column
 - If no results yet saved
 - Save the test results to file
 - Set the value (perhaps file location) in Results Saved
 - If results are already saved,
 - Save the test results to a temp file
 - Compare current results with the previous results
 - Report any discrepancies detected
- In practice, you probably want to save alphanumeric or logic results, or not bitmaps because there are serious bitmap-equivalence problems to solve.

*With this, you
have a regression
test tool.*

*Without it, you
still have a
perfectly good test
execution tool.*

A data-driven approach - 6

- We stopped at test execution.
 - There were so many variables to play with that we didn't have time to repeat tests the program had already passed. So we did risk-based regression (retest in same areas) rather than reuse based regression (retest with same tests)
 - Of course, we still had tests available for reuse to verify a bug fix
- The execution engine allowed testers to write logical specifications of their tests, focusing on creating interesting patterns of test inputs, instead of being distracted by time-consuming and error-prone test execution.
- The printouts—in this case—adequately described expected results, supporting visual inspection for pass/fail.

In essence, the tool supported exploratory testing of a complex product by a skilled tester.

Data driven architecture: What did we achieve?

- Testers used a test design tool (spreadsheet with appropriate columns) and their input to the tool *directly* translated to test execution
 - Natural interface
 - Highly resilient in the face of constant change
 - We automated execution, not evaluation
 - Testers focused on design and results, not execution
 - Saved SOME time
 - We didn't run tests twice
 - Served, in this case, as a support tool for exploration, rather than regression
 - Extends naturally to regression in appropriate cases

When we think of it as computer-assisted testing, instead of test automation, we realize that an incomplete solution can be very useful—and much cheaper than a “complete” solution.

Data driven architecture: What did we NOT achieve?

- Several problems with this approach
 - No model for generating tests. It takes whatever we provide, good or bad.
 - No model or evaluation of any type of coverage.
 - No provision for sequential effects, this is pure no-sequential combination testing.
 - We have a column for every variable, even though many tests will set far fewer variables.
 - The spreadsheet can get unmanageably wide.
 - For programs that let you set the same variable (do the same thing) repeatedly, this approach becomes unusable.

Other data-driven architectures

While we were developing this in 1992-93, Hans Buwalda was publishing a more general solution:

- Rather than put every variable into its own separate column and then define a separate setter method for every variable
- Hans would create domain-specific languages for his clients
 - The verbs are **action words** – keywords that represent a client-meaningful task and are named in a client-meaningful way
 - The nouns are data (parameters)
 - An action word might get some data, set some data, **and** manipulate some data.

The client can write her own tests by listing action words and parameters in a spreadsheet. They are read and executed via an interpreter, with the same maintainability benefits as in the last example.

Data-driven architectures aka GUI Testing Frameworks

- Since Buwalda's paper (and early papers from several others, including Bret Pettichord), this solution has gained much popularity
- All of the frameworks are code libraries that separate designed tests from code details.
 - modular programming of tests
 - reuse components
 - hide design evolution of UI or tool commands
 - independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
 - provide opportunity to routinely incorporate important utilities in your tests, such as memory check, error recovery, or screen snapshots



But many
vendors still
promote, and
many people still
try to use,
capture-replay.
After all, who
needs
maintainability,
right?

Data-driven / keyword driven approaches: common problems

- No model for generating tests
- No measurement or model of coverage
- Domain-specific languages are poorly researched and hard to design
 - The people who will be designing them in the field are typically, in terms of language design, amateurs.
- Some tools require close collaboration between business analyst (non-programming tester) and a programmer.
 - Tests may be flexible in terms of data values, but inflexible in terms of order or combination with new variables or tasks.



*Some of
these do a
better job of
supporting
exploration
than others.*

GUI Regression Automation Readings

- Chris Agruss, Automating Software Installation Testing
- Tom Arnold, Visual Test 6 Bible
- James Bach, Test Automation Snake Oil
- Hans Buwalda, Testing Using Action Words
- Hans Buwalda, Automated testing with Action Words: Abandoning Record & Playback
- Elisabeth Hendrickson, The Difference between Test Automation Failure and Success
- Mark Fewster & Dorothy Graham, Software Test Automation
- Linda Hayes, The Automated Testing Handbook
- Doug Hoffman, Test Automation course notes
- Cem Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing
- Cem Kaner, Architectures of Test Automation
- John Kent, Advanced Automated Testing Architectures
- Bret Pettichord, Success with Test Automation
- Bret Pettichord, Seven Steps to Test Automation Success
- Keith Zambelich, Totally Data-Driven Automated Testing

ACKNOWLEDGEMENT

Much of the material in this section was developed or polished during the meetings of the Los Altos Workshop on Software Testing (LAWST). See the paper, "Avoiding Shelfware" for lists of the attendees and a description of the LAWST projects.

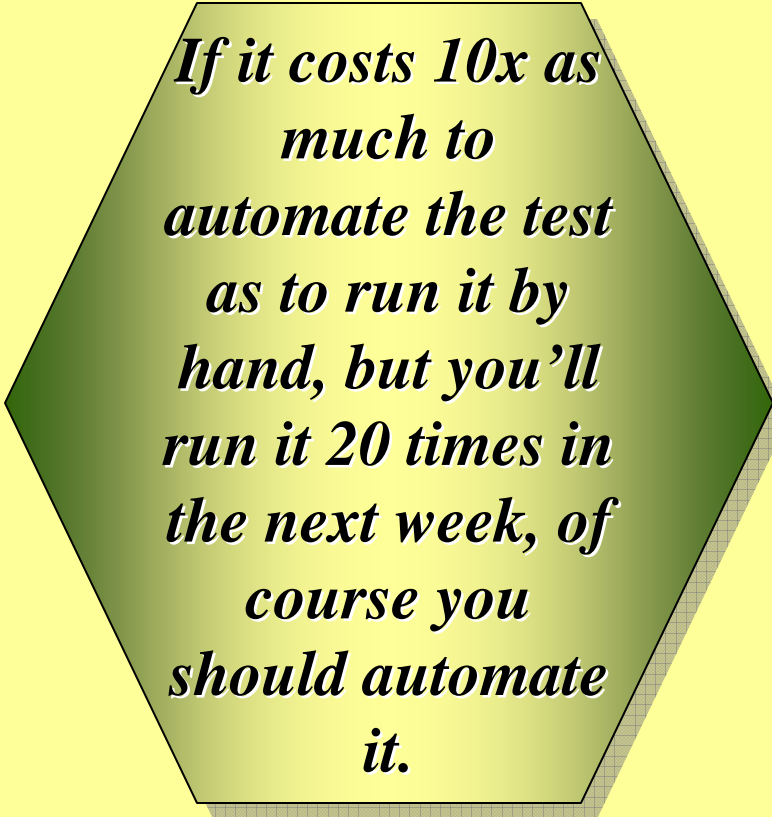
Three modern examples of data-driven approaches

- SAFS: Software Automation Framework Support
 - <http://safsdev.sourceforge.net/Default.htm>
- TestArchitect (from LogiGear)
 - <http://www.logigear.com/products/testarchitect/>
 - This is Buwalda's latest work
- FIT: Framework for Integrated Test
 - <http://c2.com/cgi/wiki?FrameworkForIntegratedTest>
- For more links to (especially open source) GUI-level and code-level test harnesses
 - <http://c2.com/cgi/wiki?TestingFramework>
 - http://www.io.com/~wazmo/blog/archives/2004_01.html
- The main GUI test tool vendors also support data-driven testing

Near-term Benefits Can be Achieved

You can plan for near-term ROI

- Some tests are fundamentally repetitive, and are reused many times in a short time
 - Smoke testing
 - Check every build with such basic tests that failure disqualifies the build.
 - Variations on a theme
 - Many instances of almost the same test, slight variations in data or timing
 - Useful for troubleshooting
 - Configuration testing
 - Test 50 printers, same test suite, same night.
 - Who would want to run this suite 50x by hand?



If it costs 10x as much to automate the test as to run it by hand, but you'll run it 20 times in the next week, of course you should automate it.

You can plan for near-term ROI

- Some tests are too hard to do manually.
Automate these tests to extend your reach
 - Load and stress testing
 - Life testing
 - Function equivalence testing
 - Performance benchmarking
 - Oracle (early 1980's) did performance comparisons of features from build to build to expose anomalous timing differences:
 - often caused by delayed-fuse bugs, like wild pointers
 - bugs that might not become visible in normal testing until much later (and then they are irreproducible).

The value of automating these is not that you save a few nickels.

The value is that automation lets you gain information that you couldn't otherwise gain.

30 Common Mistakes in Regression Automation

Common mistakes in GUI test automation

1. Don't write simplistic test cases.
2. Don't make the code machine-specific.
3. Don't automate bad tests.
4. Don't create test scripts that won't be easy to maintain over the long term.
5. Avoid complex logic in your test scripts.
6. Don't mix test generation and test execution.
7. Don't deal unthinkingly with ancestral code.
8. Don't forget to retire outdated or redundant regression tests.

Common mistakes in GUI test automation

9. Don't spend so much time and effort on regression testing.
10. Don't stop asking what bugs you *aren't* finding while you automate tests.
11. Don't use capture/replay to create tests.
12. Don't write isolated scripts in your spare time.
13. Don't assume your test tool's code is reliable or unlikely to change.
14. Don't put up with bugs and bad support for the test tool.
15. Don't "forget" to document your work.
16. Don't fail to treat this as a genuine programming project.

Common mistakes in GUI test automation

17. Don't insist that all your testers (or all the testers you consider skilled) be programmers.
18. Don't give the high-skill work to outsiders.
19. Don't underestimate the need for staff training.
20. Don't use automators who don't understand testing (or use them cautiously).
21. Don't use automators who don't respect testing.
22. Don't mandate "100% automation."

Common mistakes in GUI test automation

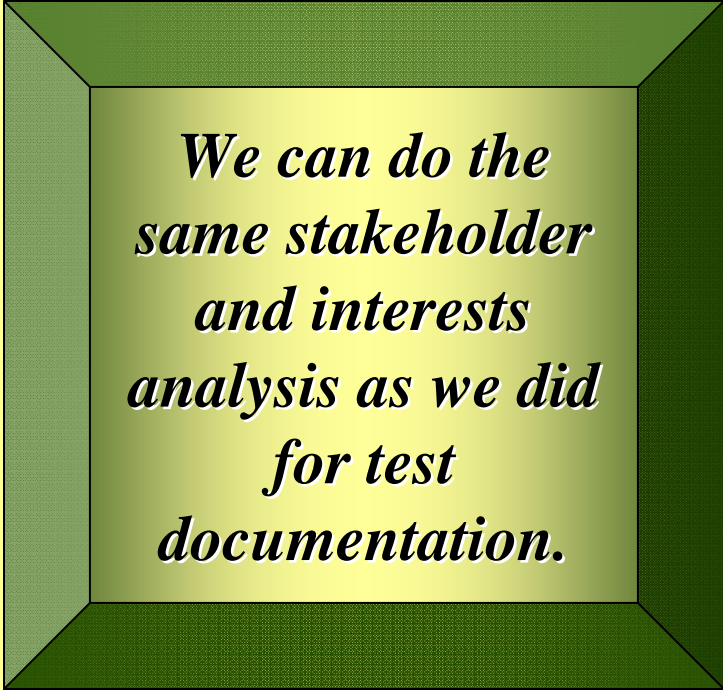
23. Don't underestimate the cost of automation.
24. Don't estimate the value of a test in terms of how often you run it.
25. Don't equate manual and automated testing.
26. Don't underestimate the need for staff training.
27. Don't expect to be more productive over the short term.
28. Don't put off finding bugs in order to write test cases.
29. Don't expect to find most of your bugs with regression tests.
30. Don't forget to clear up the fantasies that have been spoon-fed to your management.

Requirements Analysis for Regression Automation

Requirements analysis

Automation requirements aren't **only** about the software under test and its risks. To understand what we're up to, we have to understand:

- The software under test and its risks
- How people will use the software
- What environments the software runs under and their associated risks
- What tools are available in this environment and their capabilities
- The development strategy and timeframe for the software under test
- The regulatory / required recordkeeping environment
- The attitudes and interests of test group management.
- The overall organizational situation



*We can do the
same stakeholder
and interests
analysis as we did
for test
documentation.*

Requirements analysis

- Requirement:
 - “Anything that drives design choices.”
- The paper (Avoiding Shelfware) lists 27 questions. For example,
 - **Will the user interface of the application be stable or not?**
- Let's analyze it.
 - The reality is that, in many companies, the UI changes late.
 - Suppose we're in an extreme case, the UI changes frequently and very late.
 - Does that mean we cannot automate cost effectively?
 - No. It means that we should
 - Do only those types of automation that can yield a fast return on investment, or
 - Invest carefully in an approach that maximizes maintainability.

Requirements questions

- Will the user interface of the application be stable or not?
- Who wants these tests? To what degree are they favored stakeholders? What influence should they have over your test design?
- Does your management expect to recover its investment in automation within a certain period of time? How long is that period. How easily can you influence these expectations?
- Are you testing your own company's code or the code of a client? Does the client want (is the client willing to pay for) reusable test cases or will it be satisfied with bug reports and status reports?

Requirements questions

- Do you expect this product to sell through multiple versions?
- Do you anticipate that the product will be stable when released, or do you expect to have to test Release N.01, N.02, N.03 and other patch releases on an urgent basis after shipment?
- Do you anticipate that the product will be translated to other languages? Will it be recompiled or relinked after translation (do you need to do a full test of the program after translation)? How many translations and localizations?
- Does your company make several products that can be tested in similar ways? Is there an opportunity for amortizing the cost of tool development across several projects?

Requirements questions

- How varied are the configurations (combinations of operating system version, hardware, and drivers) in your market? (To what extent do you need to test compatibility with them?)
- What level of source control has been applied to the code under test? To what extent can old, defective code accidentally come back into a build?
- How frequently do you receive new builds of the software?
- Are new builds well tested (integration tests) by the developers before they get to the tester?
- To what extent have the programming staff used custom controls?

Requirements questions

- How likely is it that the next version of your testing tool will have changes in its command syntax and command set?
- What are the logging/reporting capabilities of your tool? Do you have to build these in?
- To what extent does the tool make it easy for you to recover from errors (in the product under test), prepare the product for further testing, and re-synchronize the product and the test (get them operating at the same state in the same program).
- In general, what kind of functionality will you have to add to the tool to make it usable?

Requirements questions

- Is your company subject to a regulatory requirement that test cases be demonstrable?
- Will you have to be able to trace test cases back to customer requirements and to show that each requirement has associated test cases?
- Is your company subject to audits or inspections by organizations that prefer to see extensive regression testing?
- Is your company subject to a litigation risk that you should manage partially by making sure that test cases are demonstrable?

Requirements questions

- If you are doing custom programming, is there a contract that specifies the acceptance tests? Can you automate these and use them as regression tests?
- What are the skills of your current staff?
- Must it be possible for non-programmers to create automated test cases?
- Are cooperative programming team members available to provide automation support such as event logs, more unique or informative error messages, and hooks for making function calls below the UI level?

Requirements questions

- What kinds of tests are really hard in your application? How would automation make these tests easier to conduct?
- To what extent are oracles available?
- To what extent are you looking for delayed-fuse bugs (memory leaks, wild pointers, etc.)?

Think about:

- Regression automation is expensive and can be inefficient.
- We are doing computer-assisted testing, not full automation.
- Regression is just one target of (partial) automation. You can create and run new tests instead of reusing old tests.
- Developing programmed tests is software development.
- Maintainability is essential.
- Extending your reach may be more valuable than repeatedly reaching for the same things.
- Design to your requirements.



And set management expectations with care.

In this respect, tool vendors can sometimes be your worst enemies.