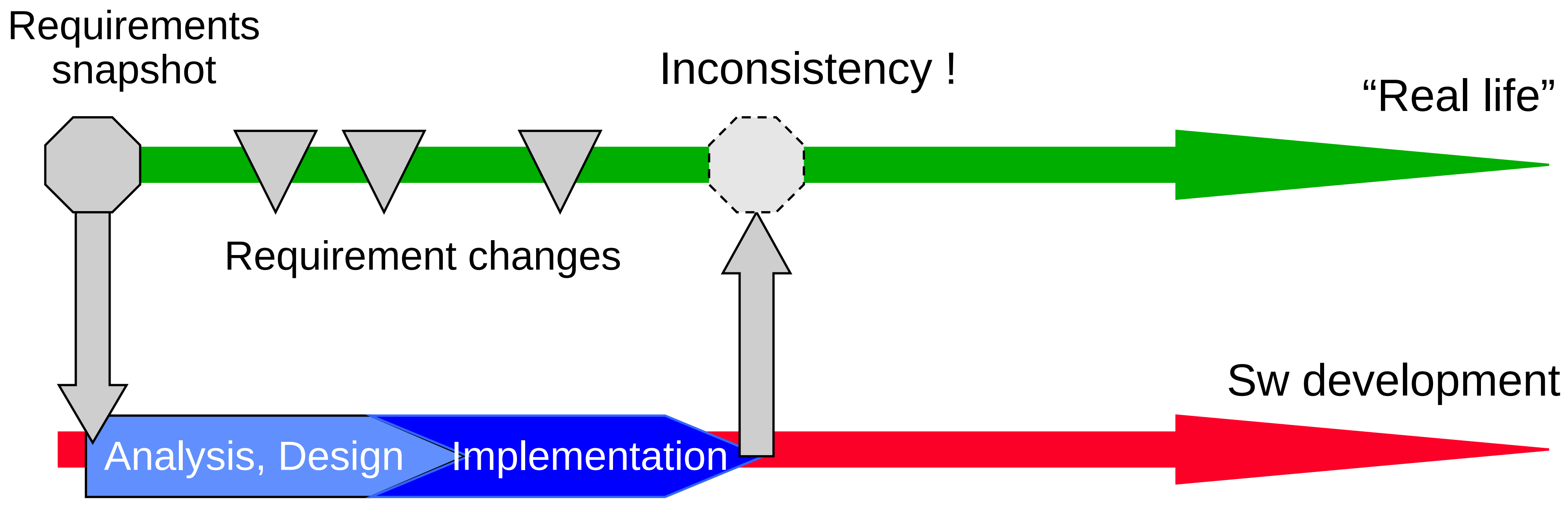
2 Development process for multimedia projects

Literature:

- Kent Beck: Extreme Programming Explained – Embrace Change, Addison-Wesley 2000
- Mary & Tom Poppendieck: Lean Software Development – An Agile Toolkit, Addison-Wesley 2003
- Matt Stephens, Doug Rosenberg: Extreme Programming Refactored: The Case Against XP, Apress 2003

Changing Requirements

- Key problem in software development
  - Requirements change during course of project



Specific drivers for requirement changes in multimedia projects:

- New technologies & devices, new (corporate) design rules, new services, …
- Feedback from non-technical reviewers (designers, customers)

Cost of Change

K. Beck, Extreme Programming Explained, p. 21 ff:

“I can remember sitting in a big linoleum-floored classroom as a college junior and seeing the professor draw on the board the curve found in Figure 1.” ...

“The software engineering community has spent enormous resources in recent decades trying to reduce the cost of change – better languages, better database technology, better programming practices, better environments and tools, new notations. What would we do if all that investment paid off?””

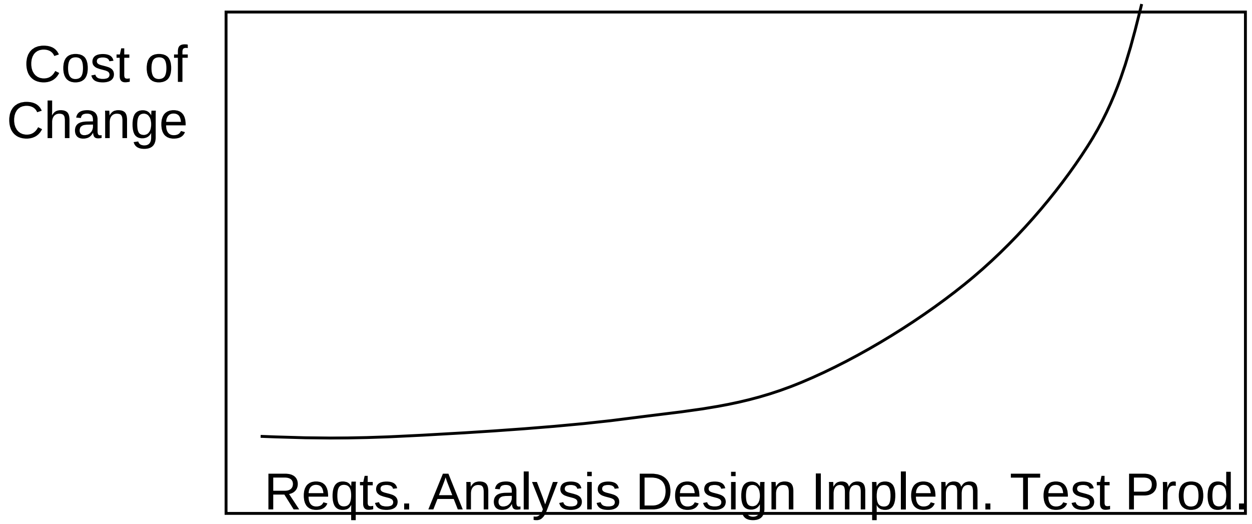


Fig. 1

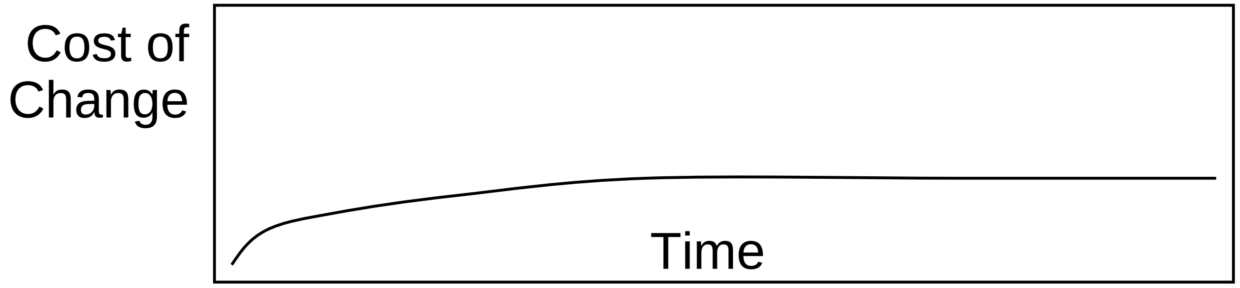


Fig. 2

“What if tomorrow’s software engineering professor draws Figure 2 on the board?”

Rationale for Agile Development

- Pace of change makes waterfall-like development processes difficult
  - Development project has to be able to react at all times to changes:
    - “*agile:* able to move quickly and easily” (Webster’s New Dictionary)
- Industrial development process standards create large overhead
  - “Travel light” (Kent Beck)
  - “Lean development”, “lightweight methods”
- Important activities are not liked by developers in traditional approaches
  - Specification, documentation, quality assurance
- Tendency towards small, short-lived projects
  - The whole industry gets more agile (short product development cycles)
- Advances in technology
  - Object-oriented programming
  - Frameworks & patterns
  - Automated testing
  - Integration of documentation and code

Origins of “Lean Thinking”

- Late 1940s, Taiichi Ohno, Toyota Corp.:
  - How to produce cars in small quantities as inexpensive as mass-produced cars?
  - Fundamental lean principle: *Eliminate waste*.
    Anything that does not create value for the customer is waste.
    - The Seven Wastes of Manufacturing (Shigeo Shingo):
      Inventory, extra processing, overproduction, transportation, waiting, motion, defects
  - Transfer from production process onto development process:
    - How much value to the customer is delivered by an ongoing development project, a design, a prototype?
- James Womack, Daniel Jones, 1996: *Value Stream Mapping*
  - Which part of the production time is spent in actually adding value, which part is waste?
    - Production of a Coca-Cola can: 319 days from mine to consumption, 3 hours spent on production, i.e. 0.04%

The Seven Wastes of Software Development

- Partially done work
  - Requirements, design documents, not yet integrated software modules
  - Idea: Try to get it into working state immediately
- Extra Processes
  - Idea: Remove all unnecessary paperwork
- Extra Features
  - Are they really needed in the end?
  - Idea: Resist the temptation, keep it simple.
- Task Switching
  - Idea: Assign a developer to one and only one project
- Waiting (for decisions mainly)
  - Idea: Try to delay decisions as long as possible and keep working
- Motion
  - Idea: Keep developers closely together, enable informal communication, try to keep a customer representative on the project team
- Defects

eXtreme Programming (XP) and Other Agile Methods

- Most famous authors promoting agile methods:
  Kent Beck, Alistair Cockburn
- Various methods, covered under the umbrella “agile”:
  - Extreme Programming (Kent Beck, ca. 1998), Crystal, Scrum, Adaptive Software Development
- Common to all agile methods:
  - Evolutionary development
  - Specific techniques of communication
- Extreme “marketing” effort – lots of “hype”
  - Currently more than 20 books
  - Only little proper evidence for success

- In the following: “Original” XP according to Kent Beck
  - With some minor comments from the discussion in the literature

The XP Team

- Small group, at most 12 persons
- Group should fit into a single room
  - Ideally the group is co-located for the whole working time
- Special member roles:
  - Representatives of the customer ("on-site customer")
  - *Tester:* Helps the customer to translate his stories into functional tests
  - *Coach:* Helps in keeping the XP discipline
  - *Tracker*: Continuously measures progress of the team and publishes it
  - *Consultant:* External provider of specific knowledge, active on the team only for a short time
  - *Big Boss:* Encourages the team

Four Values of XP

- Communication
  - XP aims to keep the right communication flowing
  - XP defines practices that require communication
- Simplicity
  - *What is the simplest thing that could possibly work?*
  - Do a simple thing today and pay a little more tomorrow to change it if needed
  - Do not do a complicated thing today that may never be used anyway
- Feedback
  - *Don’t ask me, ask the system! Have you written a test case for that yet?*
  - Minute-to minute scale feedback:
    - » Feedback about system state due to extensive automated testing
  - Week-to-month scale feedback:
    - » Customers get updates about the growth of the system
- Courage
  - Fix flaws, don’t circumvent them. Throw code away if it is unstable.

A Development Episode (1)

- A looks at his stack of task cards. The top card says "Export Quarter-to-date Withholding". He addresses B.
- "Hi B, at this morning's standup meeting I heard you had finished the quarter-to-date calculation. Do you have time to help me with the export?" B agrees, and they are a pair now for some time.
- A and B need some additional information about the data structure and interrupt C for 30 seconds. They get the answer immediately.
- By looking at the existing export *test cases*, a simple abstraction (introduction of a superclass) is found by A and B. The code is restructured (refactored) with the superclass. All existing test cases are run successfully. Some other test cases may also profit from the new superclass; this is written down on the pair's to-do card.
- A and B together write the test case for the export function. During this, they note some ideas for the implementation on the to-do card.
- A and B run the test case, and it fails as expected, due to the missing implementation.

A Development Episode (2)

- During implementation, some other test cases come to the minds of either of the two, and are noted on the to-do card. Whoever of the two has the best ideas implements the export function and later the additional test cases, the other watches and comments. After a few iterations, all new test cases run successfully, as well as the old tests.
- What's left on the to-do-card? The two restructure (refactor) other test cases and make sure all tests run successfully.
- Look, the integration machine is free! The two go to integration machine, load their new code (tests and implementation) and run all the tests known. Strangely, a not too much related test fails. After a few minutes, it has been clarified why and all tests run successfully.
- The new version is released.

This is the whole XP lifecycle in a nutshell.

User Stories

- “Each user story is a short description of the behaviour of the system, from the point of view of the user of the system. In XP, the system is specified entirely through user stories.”
  (R. Jeffries, A. Anderson, C. Hendrickson, XP Installed, Addison-Wesley 2001, p. 24)
- Preparation:
  – Everybody (representing the customer) gets a card, tries to scribble on it and tears up the card.
- Process:
  – Customer writes story on card, possibly in iterations
  – Programmers “listen” - ask questions just for clarification
  – Stories are promises for conversation
- How many:
  – At least one per feature
  – One story implementable in a few days to a few weeks time

Planning in XP

- Releases:
  - One to six month period
  - Software actually delivered to customer
- Iteration:
  - One to three weeks period
  - Produces an intermediate version for the next release
- Story:
  - Closely linked to system features
  - One or several stories to be realized in one iteration

- XP suggests to plan ahead for one to two steps at most on all three levels

(Presentation based on chapter 5 of
K. Beck/ M. Fowler: Planning XP, Addison-Wesley 2001

The Twelve XP Practices

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

All practices are interwoven with each other.

The picture will gradually become more complete over 12 slides...

Practice 1: The Planning Game

- XP develops the system in an evolutionary way, so the team has to plan what the features of the next iteration will be.
- This is a game between two parties, leading to a balanced solution:
  - Business people (customer): What are the valuable features? What are the priorities? When are the features needed?
  - Technical people (programmer): How expensive is a feature to implement? What are the consequences? What is a realistic schedule?

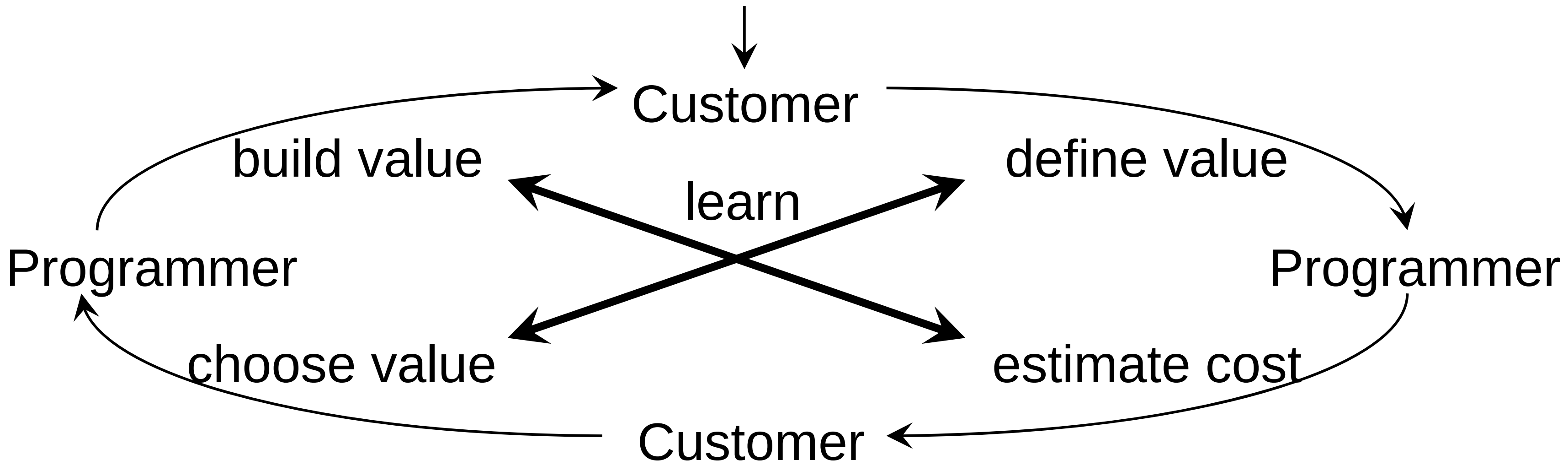


Figure from Jeffries/Anderson/Hendrickson: XP installed, Addison-Wesley 2001

Practice 2: Small Releases

- Release = Software version handed over to customer

- Every release should be as small as possible, i.e. contain a small change to the previous release
  - Containing the most valuable business requirements
- Every release has to make sense as a whole
- Reduce release cycle:
  - One month or two (if possible)

- ***Potential problems:***
  - Usually, the small releases will not go into productive use (to avoid instabilities, additional user training etc.)
  - Therefore, specific quality assessment by customer is expected
    - » Is this realistic?

Practice 3: Metaphor

- Single overarching metaphor
  - Naive: Terms mirroring the real world, e.g. contract, customer, ...
  - Less naive: e.g. pension calculation as a spreadsheet with rows and columns
- “The metaphor in XP replaces much of what other people call ‘architecture’.” (K. Beck, XP Explained, p. 56)
- In practice more similar to a domain model as done in system analysis.

- ***Potential problems:***
  - Metaphor may become too complex to be helpful for large systems. (Amr Elssamadisy, quoted in Stephens/Rosenberg p. 318)
- ***Consequence:***
  - UML Class diagram of the problem domain and its metaphor may be helpful

Practice 4: Simple Design

- "The right design for the software at any given time is the one that
  1. Runs all the tests
  2. Has no duplicated logic
  3. States every intention important to the programmers
  4. Has the fewest possible classes and methods."

  (K.Beck, p.57)

- XP mantras: "The simplest thing that could possibly work.", "You ain't going to need it. (YAGNI)"
  – Erase (or better do not add in the first place) everything unnecessary
- Software design seen as a communication medium
  – Very similar to a traditional design specification...
- Design is represented in code
  – There is no separate documentation
  – Graphical sketches (e.g. UML) only used for short digression (essentially for finding the right questions)

Practice 5: Testing

- “Any program feature without an automated test simply does not exist.” (K. Beck, p. 57)
- Test-First approach:
  - Tests are written before the program
  - Tests are used to clarify the usage of an interface, to define the expected effect, to single out problematic special cases, ...
  - Tests are the XP replacement for traditional software specification
- Automated tests:
  - Tests are kept in an executable infrastructure and can be run at any time
  - Tests give feedback and confidence to the programmer
  - “Test infected: Programmers love testing” (E. Gamma about the xUnit testing framework)
- Programmer-written unit tests: Must always run to 100%
- Customer-written functional tests based on “stories”: May run only partially for some time

Practice 6: Refactoring

- Adding new features in an arbitrary way will lead to ill-structured code
- When adding a new feature, the structure of the system may need to be adapted *(refactored)*
  - Refactoring may remove code, introduce new code but keeps the functionality unchanged (all tests run 100%)
  - Simple steps like combining parameters of a method into a data structure
  - Complex steps like applying design patterns
- This is done only when necessary to keep the solution simple:
  - Main reason for refactoring: To avoid duplication of logic
  - Possibly other reason: Smaller and more elegant design after introduction of new features
- How can we be sure not to destroy working functionality by refactoring?
  - Use automated tests
  - Refactor first, run the tests, then add the feature, run the enhanced tests

Practice 7: Pair Programming

- "All production code is written with two people looking at one machine, with one keyboard and one mouse." (K. Beck, p. 58)
- Two roles:
  - Keyboard/mouse owner: Thinks tactically about the best way to implement the method under development
  - Observer: Thinks strategically about the overall approach and simplification
  - Roles in the pair may be switched after some time
- Dynamic pairing:
  - Pair partners should change frequently
  - Changes may take place even several times a day

- ***Potential problems:***
  - Complex programming tasks are often better solved by a single person "meditating" over the solution
  - Some people simply do not like the pair situation

Practice 8: Collective Ownership

- Through pair programming, any piece of code has many authors
  - Side effect: Removes too complex code
- *Everybody in the team has the right to add value to any portion of the code at any time.*
- If somebody does not know some part of the system well, he/she should pair up with an expert on this part
- Practical tool for co-ordination: *Stand-Up Meetings*
  - Regular meetings in the morning of each day
  - Everybody has to *stand* in a circle (to keep the meeting short)
  - In turn, each team member reports what has been done yesterday and what the plans are for today

- ***Potential problems:***
  - Collective ownership means no individual responsibility
  - Collectivism partially contradicts to human nature

Practice 9: Continuous Integration

- There is always an up-to-date running version of the full system.
- Integration is not a late development stage but done on a daily basis.
- Once a day, newly developed code is integrated into the common code basis
- Separate “integration machine”
  - Can be used only by one developer pair at a time
  - Keeps the current version of the integrated system
  - Programmers
    - » load new/modified code
    - » check for collisions
    - » run all tests, fixes problems, ... until all tests run 100%

Practice 10: 40-Hour Week

- Overtime considered a symptom for a serious problem of the project
- XP-rule: *You can’t work a second week of overtime.*
- Basic idea: People should not get too exhausted

- ***Potential problems:***
  - Customers may have the feeling the team does not work hard enough
  - Fully unrealistic when delays occur and important deadlines are approaching
  - The on-site customer is a single point of failure who has low chances for a 40-hour week

Practice 11: On-Site Customer

- "A real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities. By "real customer" I mean someone who will really use the system when it is in production." (K. Beck, p. 60)
- Concept of on-site customer has developed since the original book of 2000
  - Now "Whole team": There may be several representatives of the customer
  - Only representative of the customer required, no longer a "real customer"

- ***Potential problems:***
  - Definitely the weakest point of the XP approach.
  - On-site customer tends to get overloaded, since he/she has the final responsibility for too many things
  - Customer organisations tend to assign the role to inexperienced people
  - On-site customer gets mentally separated from his organization and may feel too much as team member to be effective.

Practice 12: Coding Standards

- To enable collective ownership, coding standards for the team are necessary
  - Naming conventions
  - Conventions for embedded documentation
  - Code layout conventions
  - Framework for automated tests
- These standards need to be discussed thoroughly, so that everybody accepts them without resistance
- Fortunately, for modern languages “style guides” often exist already.

Elements of XP and their Applicability to Multimedia Authoring

• The Planning Game	applicable directly
• Small releases	applicable directly
• Metaphor	applicable in adapted way
• Simple design	What is simple design in e.g. Flash?
• Testing	How to automate tests for Flash?
• Refactoring	applicable in adapted way
• Pair programming	applicable directly
• Collective code ownership	applicable directly
• Continuous integration	applicable directly
• 40-hour week	applicable directly
• On-site customer	applicable directly
• Coding standards	applicable / which standards?

Simple Design: Trade-Offs in Multimedia Authoring

- The following trade-offs are obvious with Flash, but exist in some form in any multimedia authoring tool.
- **Design vs. Behaviour** trade-off
  - Shall we work out the (graphical/sound) design first, or shall we try to understand the full interaction structure first?
- **Scripting/architecting** trade-off
  - Shall we simply start to spread scripting code over the animation symbols, or shall we try to design a scripting architecture first?
  - Examples for scripting architectures:
    - » MVC (e.g. Account example)
    - » Only global code (e.g. Sound example)
    - » All code in external classes (e.g. moving ball example)
- **Instance/schema** trade-off
  - Shall we simply assign behaviour to instances, or shall we bother to define generic behaviour and customization for the instances?

Extreme Multimedia Authoring: What is it?

- Here are some suggestions (to be verified during projects):
- Start from the graphical design, but integrate simple behaviour soon.
  - Study alternatives to find the most natural way of representing the system.
  - Refine later (design and behaviour)
- Use the ***simplest possible scripting*** approach first.
  - Keeping code on the main timeline is not object-oriented, but often helpful for experiments
- Refactor ***when necessary***:
  - Complex business logic with multiple views: Introduce MVC architecture
  - Multiple instances of a symbol with generic behaviour:
    - » Attach code to symbols and their instances
    - » Consider usage of linked ActionScript classes
  - Refactoring is easy: Code snippets can be moved around
- Create a testing infrastructure and archive tests
  - Always test full functionality after refactoring
- Work in ***short evolution cycles!***

Unit-Testing ActionScript Classes

- Principle:
  - Test cases written as ActionScript code
  - Conventions (and test framework) for systematic execution of tests
- Test frameworks for unit tests:
  - Well-known standard: derivations of xUnit (e.g. JUnit)
  - *ASUnit* test framework for ActionScript available on the Web
    - » see htttp://www.asunit.com
  - Simple “framework” can be written with little effort
    - » see next slides for a simple test infrastructure
- Limitations:
  - Does work only with pure ActionScript classes
  - Not applicable to graphical input/output and timeline animations

Simple Test Infrastructure: Superclass Test

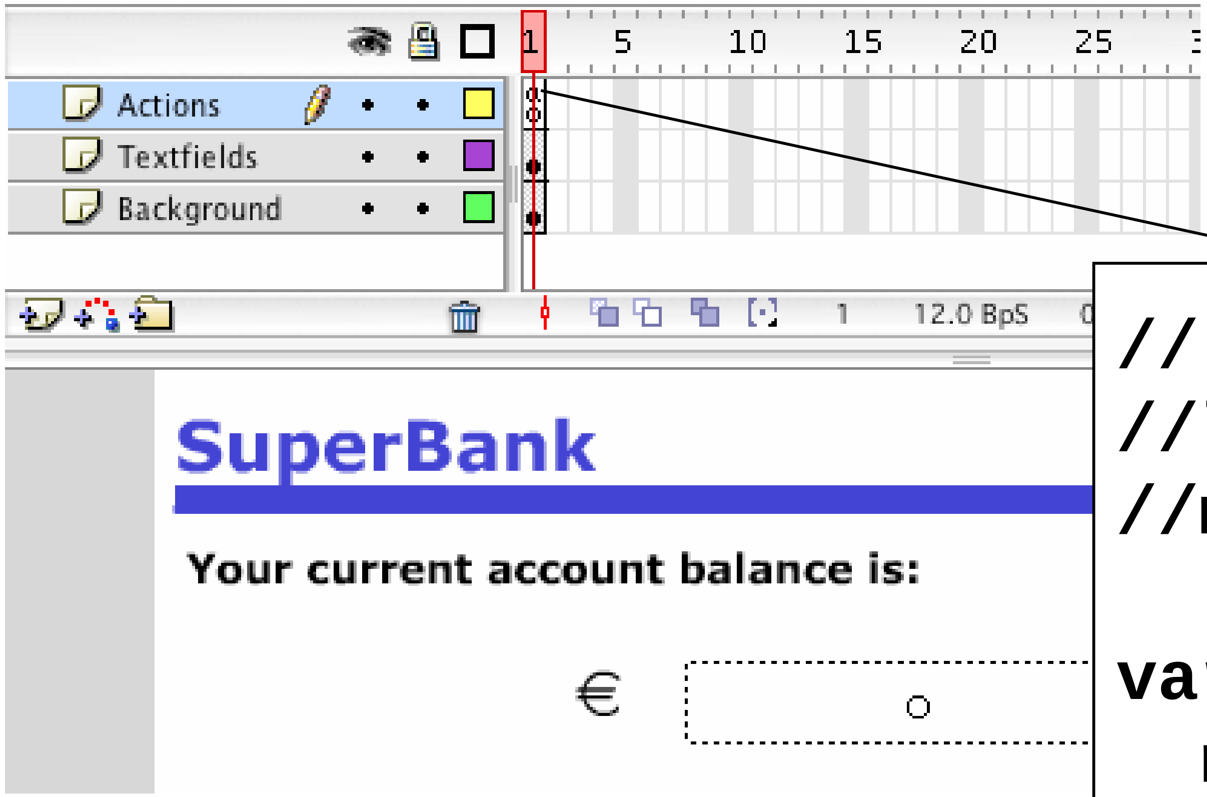
```
class Test {

  var success:Boolean = true;

  function setUp() {};       //abstract
  function classTest() {};  //abstract

  function assertEqualNumber(x:Number, y:Number) {
      if (not(x == y))
            success = false;
  }

  function runTest() {
      setUp();
      classTest();
      if (success)
            trace("Test successful!");
      else
            trace("Sorry, test failed.")
  }

}
```

Design pattern applied: “Template Method” (Gamma et al.)

Simple Test: Class AccountTest

```
class AccountTest extends Test {

  var acc1, acc2:Account;

  function setUp() {
      acc1 = new Account(123);
      acc2 = new Account(234);
  }

  function classTest() {
      var amount:Number = 100;
      acc1.debit(amount);
      acc2.credit(amount);
      acc1.credit(amount);
      acc2.debit(amount);
      assertEqualNumber(acc1.getSaldo(),0);
      assertEqualNumber(acc2.getSaldo(),0);
  }

}
```

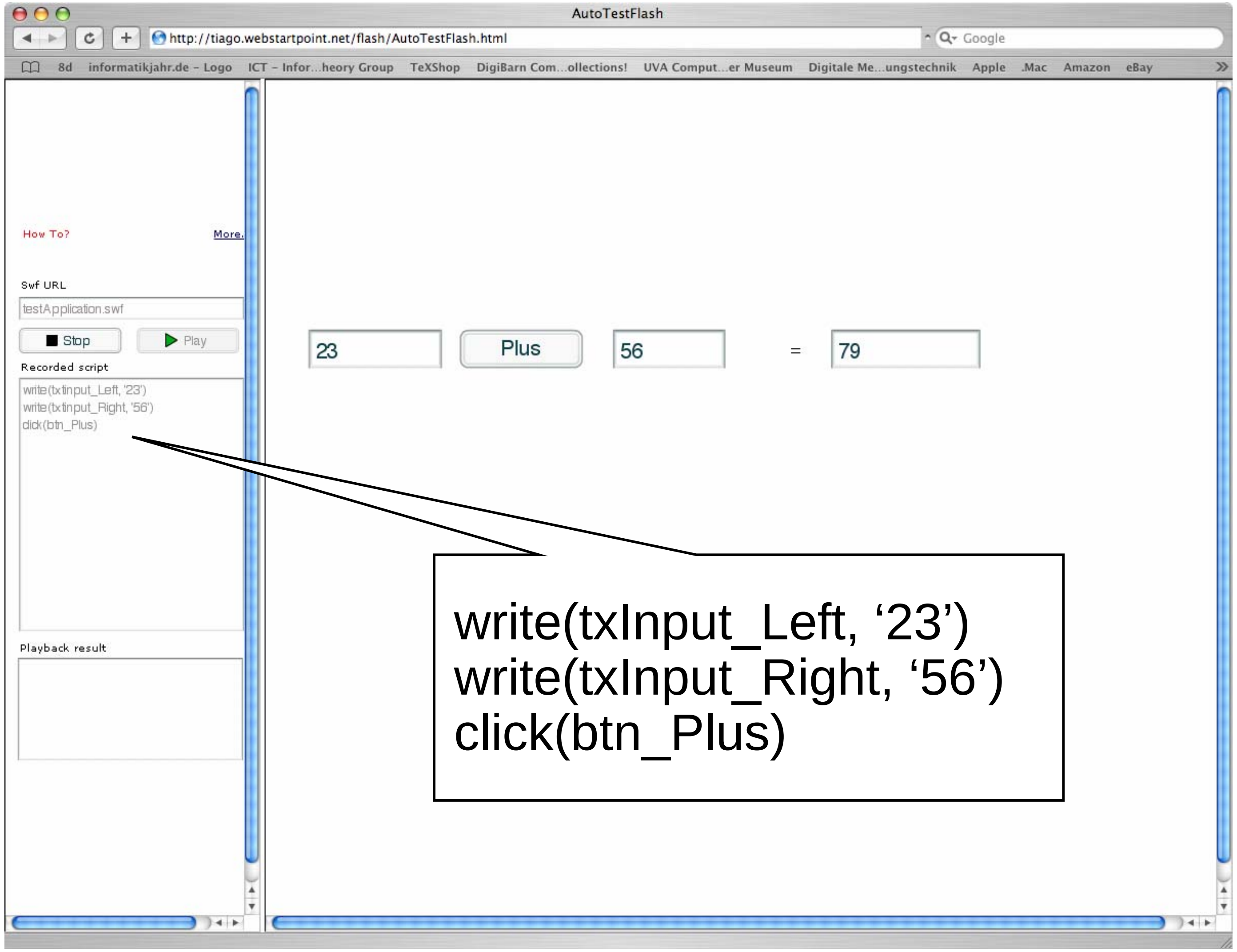
Simple Test: Make It Mandatory



```
// First of all,
//let's test the
//model classes

var at =
  new AccountTest();
at.runTest();

...
```

UI-Testing with Flash

- Tests for user stories should be based on interaction
- Tools for UI testing:
  - Record/playback facility for interface events
  - Ideally based on UI elements, not just on pixels
  - E.g. *AutoTestFlash*, see http://osflash.org/autotestflash


AutoTestFlash
http://tiago.webstartpoint.net/flash/AutoTestFlash.html
Google
8d informatikjahr.de – Logo ICT – Infor...heory Group TeXShop DigiBarn Com...ollections! UVA Comput...er Museum Digitale Me...ungstechnik Apple .Mac Amazon eBay
How To?
More.
Swf URL
testApplication.swf
Stop
Play
Recorded script
Playback result
23
Plus
56
=
79

```
write(txInput_Left, '23')
write(txInput_Right, '56')
click(btn_Plus)
```

write(txInput_Left, ‘23’)
write(txInput_Right, ‘56’)
click(btn_Plus)