Agile Development

Hands-on Construction Workshop with Boost

Kevlin Henney

kevlin@curbralan.com

Agenda

- Intent
  - Put some agile techniques into practice on a library-centric example using Boost libraries
- Content
  - *Sprint 0*: Conception
  - *Sprint 1*: Construction
  - *Sprint 2*: Construction
  - *Sprint 3*: Completion

BoostCon '07, Aspen, 17th May 2007.

Kevlin Henney

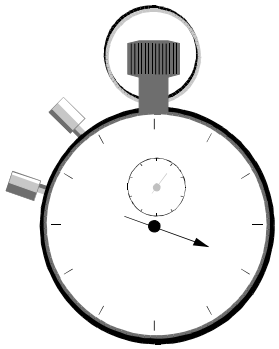
`kevlin@curbralan.com`
`kevlin@acm.org`

Curbralan Ltd

`http://www.curbralan.com`
Voice: +44 117 942 2990
Fax: +44 870 052 2289

Sprint 0: Conception

- Intent
  - Begin!
- Content
  - Introduction
  - Agile development
  - The workshop process
  - The workshop project
  - Suggested objectives



Workshop Goals

- Put agile development techniques into practice, especially incremental and TDD approaches
  - Short iterations, functionally complete increments, pairing to write C++, test-first style and tracking
- Time is short, so time management matters
  - Be realistic rather than ambitious
- However, this is not a coding contest
  - Working in a test-first style, with your team and within time are the challenges to focus on

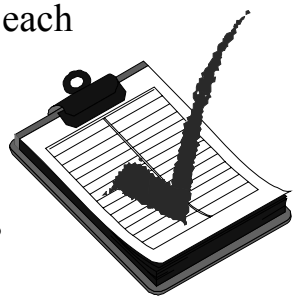
Agile development processes are intended to help developers avoid the problems of analysis paralysis, big up-front design, rushed testing and changing requirements. They treat analysis and design as continuous activities that start early in development but continue throughout, rather than as segregated phases divorced from other development activities.

Development is dynamically planned as incremental and iterative. Coding and testing are considered together and from an early stage in development. In particular, incremental design, continuous testing and responsive refactoring make up the programmer-facing discipline of Test-Driven Development (TDD). The goal of this workshop is twofold: (1) it aims to offer attendees hands-on experience of many of the practices involved in the construction phase of a development lifecycle and (2) it introduces some of the common Boost libraries to raise development above the level of standard C++ libraries.

Attendees will learn about the development side of agile development by doing it and some common Boost libraries by using them. The workshop is based on undertaking four rapid sprints of development, working on a clearly bounded and well-defined problem. The technical emphasis is on the role of libraries in promoting responsive and streamlined development, with Boost as a leading concrete example. The process emphasis is on scope management, iteration planning, TDD, pair programming and other practices and principles drawn from agile approaches such as Extreme Programming, Scrum and Lean Software Development, with guidance and feedback both during and in between iterations. A simple one-day process — the CODA (Compact One Day Agile) process — is outlined for the workshop.

Workshop Prerequisites

- You...
  - Prerequisite is a working knowledge of standard C++ and basic familiarity with Boost
- Laptops...
  - Two laptops with similar setup for each team of four
  - Easy to use IDE or editor
  - Reasonably standard C++ compiler and recent Boost libraries



The workshop is aimed at developers with competent C++ skills. Deep C++ knowledge and TDD experience are not a prerequisite, but some knowledge of agile concepts is helpful, as is some familiarity with Boost libraries.

Laptops will need to have a reasonably standard C++ compiler installed (so VC6 doesn't really cut it), a recent Boost library release and an editor that is easily usable without years of training, no matter how brilliant it might otherwise be (so, the need to master meta keys or modes is out — unless, of course, all team members agree to it).

Time management is important to the running of the workshop, so coffee breaks need to be timeboxed.

Agile Development

- So, what is agile development?
  - There are many definitions, but in the spirit of agility, it's best to keep things simple!
- Look up *agile* in the dictionary
  - That's normally clear enough should give you some idea of the objectives and values
- But beware of dogma and mythconceptions
  - E.g. *Agile* is not a process, *agile* is not a synonym for *XP*, agility is not about not doing design

Agile development can sometimes end up being little more than a buzzword, but there is more than just a superficial notion of fashion. The essence is that of rapid development with responsibility and responsiveness.

Agile Processes

- There are a number of different documented agile processes and many undocumented ones
  - E.g. Extreme Programming, Scrum, Crystal Clear, Lean Software Development, DSDM
  - Process adaptation is key to making agile development successful — adoption straight out of a book or by mandate are popular ways of failing
- Agile processes vary in philosophy, practices, degree of ceremony and lifecycle models

There are a number of useful references that describe agile philosophy, existing agile processes and aspects of agile processes:

Manifesto for Agile Software Development, `http://agilemanifesto.org`.

DSDM Consortium, `http://www.dsdm.org`.

Extreme Programming Explained: Embrace Change, 2nd edition, Kent Beck, Addison-Wesley, 2005.

XProgramming.com, `http://www.xprogramming.com`.

Agile Software Development with Scrum, Ken Schwaber and Mike Beedle, Prentice Hall, 2002.

Agile Project Management with Scrum, Ken Schwaber, Microsoft Press, 2004.

Scrum Development Process, `http://www.controlchaos.com`.

Lean Software Development, Mary Poppendieck and Tom Poppendieck, Addison-Wesley, 2003.

Organizational Patterns of Agile Software Development, James O Coplien and Neil Harrison, Prentice Hall, 2005.

Agile Techniques and Thinking

- Agile approaches are characterised by their emphasis on feedback and sustainability
  - Adaptiveness is needed in both code and practices
  - Quality of architecture and code detail matter — quality debt costs time and money in the long term
  - Design needs to be informed by sufficiency, with a strong emphasis on reducing accidental complexity
  - In contrast to sequential process models, which are *plan* driven, agile processes are *planning* driven

The following references give a sense of the tools, techniques and thinking often involved in agile development:

Slack, Tom DeMarco, Broadway Books, 2001.

Test-Driven Development: By Example, Kent Beck, Addison-Wesley, 2003.

Refactoring: Improving the Design of Existing Code, Martin Fowler, Addison-Wesley, 1999.

Refactoring Home Page, `http://www.refactoring.com`.

Working Effectively with Legacy Code, Michael C Feathers, Prentice Hall, 2005.

The Practice of Programming, Brian W Kernighan and Rob Pike, Addison-Wesley, 1999.

The Pragmatic Programmer, Andrew Hunt and David Thomas, Addison-Wesley, 2000.

The Programmer's Stone, Alan G Carter and Colston Sanger, `http://www.reciprocality.org/Reciprocality/r0`.

Structuring Use Cases with Goals, Alistair Cockburn, `http://members.aol.com/acockburn/papers/usecases.htm`.

Writing Effective Use Cases, Alistair Cockburn, Addison-Wesley, 2001.

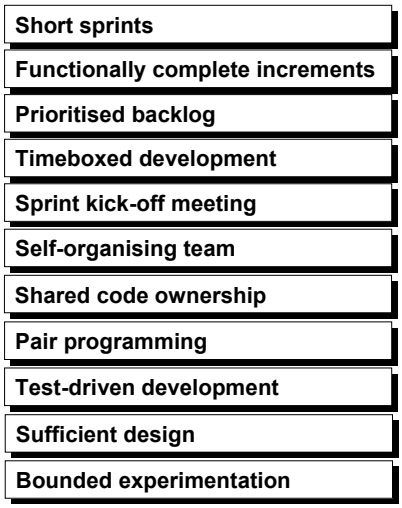
Agility and the Role of Libraries

- Libraries capture commonality as commodity
  - For software to be in a more continual state of readiness, development effort needs to be focused on work that adds value faster than it incurs overhead
  - Development velocity and readiness are improved by building on proven infrastructural parts
- Criticality of knowledge
  - Right tools for the right job used in the right way

There are many development tools that support the objectives of agile development. Often the focus is on build and environmental tools, the need for which becomes stronger the larger a development effort becomes. However, these are not the only tools in town: the stuff of code is just as open and necessary to commoditisation as the process that surrounds it. Solid libraries, with a sufficient working knowledge of them, can make a significant difference to development time, reliability and responsiveness. Layered atop the C++ standard library, the Boost libraries are one of the common and successful examples of such commodity in C++ development.

Compact One-Day Agile Process

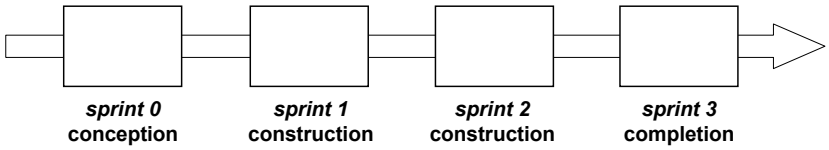
- Simple workshop process based on practices from other processes
  - Coffee breaks and lunch divide the sprints
  - Aim for functionality and test completeness
  - Focus on code quality — avoid both über-design and rabid hacking



- *Short sprints* ensure that the development steps are small enough to give frequent feedback on progress and opportunity for adjustment.
- *Functionally complete increments* guarantee that the end result of each sprint is meaningful and whole.
- *Prioritised backlog* organises the work in terms of functionality and its value.
- *Timeboxed development* ensures that development time is clearly bounded within a sprint, regardless of development status. There is more value in keeping a regular heartbeat than perpetually re-estimating and postponing runaway development.
- *Sprint kick-off meeting* ensure that each sprint begins with team communication to establish clear goals for a sprint.
- *Self-organising team* implies that the responsibilities for work allocation and for making technical decisions lies with the team.
- *Shared code ownership* places the ownership of the code with the team rather than with its individuals.
- *Pair programming* encourages learning, reviewing and shared design discussion.
- *Test-driven development* ensures that code is conceived of in terms of usage and that it is tested. A test-first style, with writing of test cases and production code interleaved, leading with test code.
- *Sufficient design* constrains the design decisions made to be sufficient for the development task at hand, curbing speculative generality and technical scope drift.
- *Bounded experimentation* is a prototyping activity intended to answer a question about feasibility or complexity of an implementation approach. It is bounded in terms of the technical scope and in terms of the time put aside for it.

The CODA Macro Process

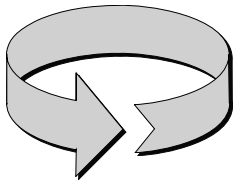
- Iterative and incremental approach
  - Begin with initial understanding of requirements plus rough up-front design...
  - Proceed through functionally complete increments...
  - Complete with at least the minimum usable subset



The lifecycle of the one-day workshop process is divided into four iterations, numbered from 0 to 3. Given the short cycle time, the term *sprint* (taken from Scrum) seems more appropriate than the more conventional *iteration*! The initial sprint is focused on familiarisation and setup. The remaining sprints are all intended to be development sprints and have a similar structure and share similar objectives. The exception is that the final sprint has a strong focus on completing any outstanding, partially completed functionality and ensuring that the minimum usable subset has been completed.

The CODA Micro Process

- With the exception of this initial sprint, each sprint is based on a test-driven approach
  - Start each sprint with a meeting to discuss issues, review progress and establish objectives and tasks
  - Write test cases and production code together, leading with test cases
  - Keep the code clean and loosely coupled
- Meet as a team, pair for coding
  - But do not stay in the same pairs



Within each development sprint, the sprint should kick off with an initial meeting to select and clarify the requirements that are to make up the increment to be developed by the end of the sprint. Feedback and observations from previous sprints should be noted during the kick-off meeting. An approach to tackling the work should be established and then pairs should tackle the work they have committed to, adopting a test-first approach. It is also worth exchanging pairing partners part way through each iteration, as well as having a short mid-sprint meeting to check on progress.

Workshop Project

- Objective...
  - Develop a C++ library for reading and writing JSON using the standard library and Boost libraries
- Background...
  - JSON (JavaScript Object Notation) is a textual serialisation format
    - *http://json.org*
    - *http://json.org/www06json.ppt*
    - *http://tools.ietf.org/html/rfc4627*

Can choose to use just plain ISO C++, ISO C++ plus the TR1 libraries (as found in Boost), ISO C++ with Boost as a whole, or any point in between.

JSON

- JSON is a minimal text-based data format
  - A declarative subset of ECMAScript (as implemented by JavaScript and JScript)
  - Simpler to work with than XML for many types of data interchange
- Simple built-in data types that can express flat and hierarchical data structures
  - Dynamic rather than statically declared data
  - No ability to define new types or anything but data

JSON is not a markup language or a full data-modelling language. It supports simple data values and record-like types made up of other values, as well as sequences of values. There is no concept of cross linking, so there can be no cyclic data structures or DAGs, only trees. JSON is only used to define data values, not data types. The type model is dynamic and, from the perspective of an application, convention based.

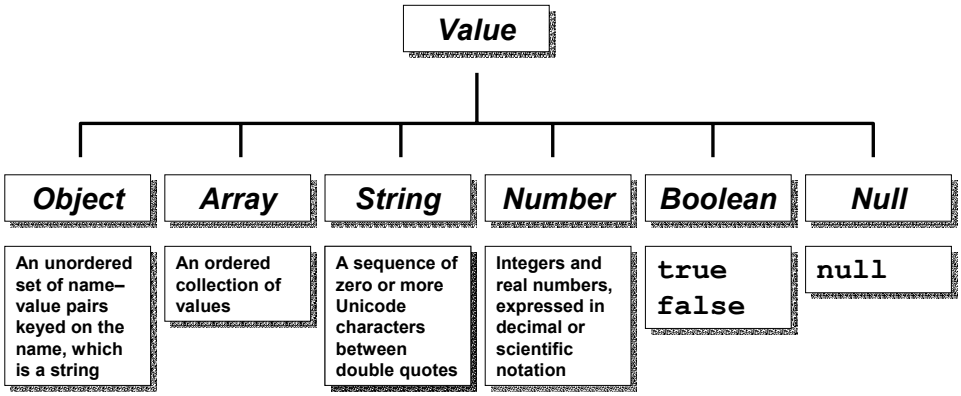
In ECMAScript the built-in `eval` function is used to convert from string form to a corresponding ECMAScript data structure. In other languages a parser of some kind is needed — one goal of this workshop is to provide such a parser for C++.

Example of JSON

```
[
    {
        "last name": "Einstein",
        "first name": "Albert",
        "does God play dice?": false,
        "date of birth": { "year": 1879, "month": 3, "day": 14 },
        "date of death": { "day": 18, "month": 4, "year": 1955 }
    },
    {
        "last name": "Bohr",
        "first name": "Niels",
        "does God play dice?": true,
        "date of birth": { "year": 1885, "month": 10, "day": 7 },
        "date of death": { "year": 1962, "month": 11, "day": 18 }
    },
    {
        "last name": "Hawking",
        "first name": "Stephen",
        "date of birth": { "year": 1942, "month": 1, "day": 8 },
        "does God play dice?": null
    }
]
```


The example above shows an array of three objects, which contain fields that variously hold strings, Booleans, nulls and further objects. It also demonstrates that not all objects that play a corresponding role have to have their keys in the same order or even contain the same keys.

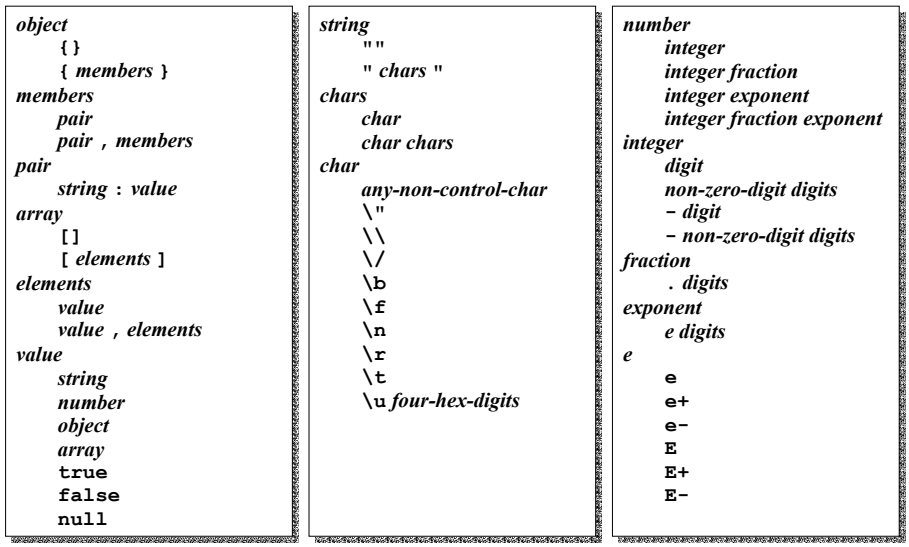
JSON Types



The data types in JSON are a subset of those found in ECMAScript. There is no concept of declarations or user-defined types, and no notion of behaviour, i.e. methods.

Although the form of numbers can be integer or real, no distinction is made in terms of type. Individual characters are expressed using strings. A null can be used to indicate the absence of a value, such as N/A for a string field or NaN for a number.

JSON Syntax



In addition to the productions shown above, there are some additional ones whose definitions are reasonably self-explanatory:

By *four-hex-digits*, four adjacent hexadecimal digits is intended, where a hexadecimal digit is a *digit* (see below) or `a`, `A`, `b`, `B`, `c`, `C`, `d`, `D`, `e`, `E`, `f` or `F`.

By *digit* one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` or `9` is intended.

By *digits* one or more occurrences of *digit* is intended.

By *non-zero-digit* one of `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` or `9` is intended.

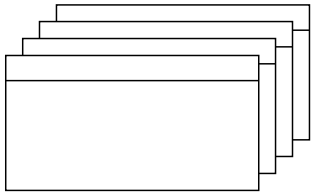
By *any-non-control-char*, any Unicode character excluding control characters and solidus characters (`\` and `/`) is intended.

Also note that spacing is allowed around punctuators (`{`, `}`, `[`, `]`, `,`, `:`).

In addition to the obvious omissions, there are a number of features supported in ECMAScript that relate to the data types described but are not supported in JSON: comments, single-quoted strings, octal and hexadecimal literals, literal forms for NaN and infinity.

Goal-Structured Requirements

- Decompose functional requirements into small task-oriented goals focused on usage
  - Identify goals that are specific, fine grained and define discrete, visible incremental steps, i.e. "read strings without spaces or escape codes" rather than "parse input"
  - Keep the focus on functionality, i.e. "read Booleans" rather than "add Boolean handling to parser"



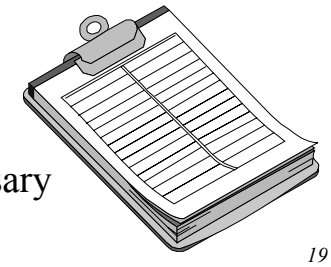
One of the simplest but most important features of an agile development is the visibility of its progress. This does not require sophisticated tools: lightweight tools and physical visualisation are considered more useful to a team than most project management tools.

Name each requirement and capture any relevant notes on an index card. Emphasise goals that correspond to functionality. There may of course be further tasks that do not fit this functional or usage-centred view, so note these requirements and tasks that are about infrastructure, logistics, etc., marking them differently, e.g. different colour cards.

In terms of progress, it is important to focus on completion of requirements, with either reduced scope or incomplete scope, than on code-centred artefacts. The worst state for a requirement to be in at the end of a sprint is 'incomplete'. Note that for a requirement to be considered complete, tests must exist to demonstrate such completion. The desire for visibility and proven completeness reinforces the emphasis on reasonably fine-grained decomposition of requirements.

Product and Sprint Backlogs

- A product backlog is the prioritised list of requirements yet to be developed
  - Includes functional and end-user requirements, such as documentation
- A sprint backlog holds the requirements for the current sprint
  - Often also includes developmentally focused items
  - Lower priority items cut as necessary



Progress can be tracked by seeing how the product backlog evolves, both through burndown charts and incremental progress.

Items in a backlog can be very fine-grained, such as distinguishing between support for plain integers, decimals and scientific notation, which allows them to be scheduled more freely, as well as providing clearer test criteria.

Prioritising Requirements

- Prioritise requirements using MoSCoW...
  - "Must have", "Should have", "Could have", "Won't have this time but would like in future"
  - "MUST" also corresponds to the "Minimum Usable SubseT"
- Development schedule needs to balance...
  - Priority (which may change over time)
  - Risk (i.e. unknowns and critical decisions)
  - Team (parallel development between pairs)

Not all requirements were created equal. For example, support for Unicode escape codes is a lower priority than support for common text and, indeed, Unicode support (assumed to be accommodated by `wchar_t`) is a lower priority than just supporting simple narrower, extended ASCII characters (assumed to be accommodated by `char`). Supporting simple strings is a higher priority than also supporting control-character escape codes. Likewise, configurability of output (i.e. pretty printing) is a lower priority than having some form of correct output, or configurability of error handling.

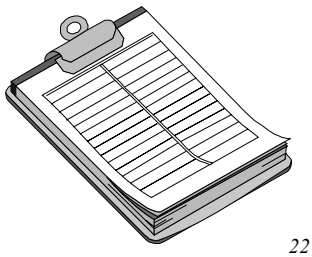
Design Suggestions

- Essentially three main parts to the solution...
  - A reader (or decoder) that interprets JSON
    - Depending on preference and design style, this can be further decomposed, e.g. lexer, parser and builder
  - A writer (or encoder) that generates JSON
  - A C++ data structure for holding data in JSON object model form
    - This can be a smart custom type or a loose protocol around existing library types

The reader may be further subdivided into lexing and parsing responsibilities. If you wish, readers and writers can be considered stream manipulators or stream extractors and inserters, alternatively a more conventional interface style can be used. Whichever style you favour, watch out for coupling implications. It is up to you whether the reader and writer are based on strings or streams. Other questions of design style include execution and organisation, such as a navigable object model versus event-driven model of traversal — essentially a pull versus push approach or an iterator and lookup versus a callback approach.

Suggested Activities for *Sprint 0*

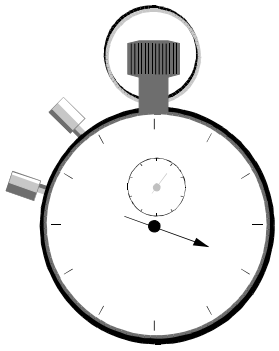
- Developmental...
  - Establish an initial and shared understanding of the domain model
  - Identify and prioritise development requirements
  - Establish an approach to the solution
  - Set up development environments
  - Agree on coding conventions
- Other...
  - Introduce yourselves



For the initial sprint the objectives are focused on familiarisation with the problem, discussing a solution approach and setting up the development environment. There may also be opportunity to experiment with a couple of ideas or unfamiliar parts of the Boost libraries that might be used in the solution.

Sprint 1: Construction

- Intent
  - Start development in earnest
- Content
  - TDD
  - *Boost.Test*
  - General guidance
  - Suggested objectives



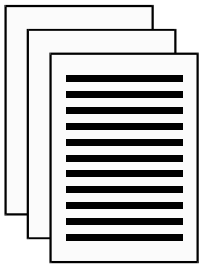
Test-Driven Development

- TDD can be characterised in terms of...
  - *Programmer testing*: test what you write
  - *Automated tests*: don't do it by hand
  - *Example-based test cases*: black-box testing of representative cases, not per-function testing
  - *Active test writing*: tests are specs, not merely verifications, that can phrase and influence design
  - *Sufficient design*: no more than is necessary
  - *Sustainable refactoring*: but allow yourself change

The emphasis of TDD is different to traditional testing approaches in that, perversely enough, it is not about testing. The notion is that a test specifies a behaviour, which means that it is a tool for expressing requirements and exploring and framing a design. As a by-product, it is also the means of verifying satisfaction of a requirement and viability of a design, i.e. it becomes an executable specification. A test-first style is encouraged for this workshop, but is not a prerequisite for using TDD in general.

Testing as Specification

- TDD is often misunderstood as, first and foremost, a testing methodology
  - Perhaps unsurprising given the name!
- A test is a proposition about some aspect of a piece of software
  - It states a requirement
  - It defines scope and, therefore, a fine-grained increment
  - It provides a means of verification



Although there is a pragmatic aspect in considering TDD from a traditional testing point of view — developers are notorious for not writing tests for their code after they've written their code, so developing tests while developing code helps to address this — TDD is not best understood from the perspective of classic testing.

A test states a requirement, as well as the means of confirming conformance to that requirement. In essence, a test can be considered an executable specification. For functional behaviour, the simplest and most readable approach to presenting functionality is by examples that illustrate the requirement on typical cases, edge cases and failure cases. For operational behaviour, such as performance, a different style of testing is needed (this workshop is focusing only on functional behaviour at the code level).

A common mistake is to focus on individual functions as the unit of test. Although there are cases where this makes sense, usage of both objects and functions is typically better characterised by defining a situation and a usage. For example, by putting an object into a particular state (e.g. "just constructed" or "just raised event *X*") and then describing what must be true of that situation, or calling an algorithm on a container with a particular set of characteristics (e.g. empty, normalised, equal values).

In an incremental approach, testing also helps to clarify what has been specified and implemented and therefore, by implication, what has not. This helps to avoid the situation of producing poor solutions with ill-defined scope — focusing on sufficient design for a particular scope is not the same as choosing the dumbest solution and forgetting about it.

Testing as Design

- TDD is also part of a design dialogue
  - First use is in the hands of the implementer, so experience of using a class, function or subsystem in a test provides feedback and clarification
- TDD tends to encourage a loosely coupled approach to design
  - Problems with coupling are signalled early
  - Mock objects provide a useful supporting technique, i.e. objects that stand-in for externals

In terms of coupling, unit tests provide a clear warning early on. When tests are difficult to write because they require large-scale environmental set up, they become cumbersome and cease to be unit tests. This is not a problem with tests, it is a problem of design that tests have highlighted.

Boost.Test

- *Boost.Test* supports many different styles of use, including lightweight unit testing
  - Header-only usage
  - Automatic registration of test cases
  - Assertions for defining and verifying expectations

```
#include <boost/test/auto_unit_test.hpp>
...
BOOST_AUTO_TEST_CASE(test_that_foo_should_bar)
{
    ... // set up test situation
    BOOST_CHECK(a necessary truth);
    BOOST_REQUIRE_EQUAL(lhs, rhs);
    ...
}
...
```


A simple usage and set up for *Boost.Test* is as follows:

- Define a separate `.cpp` test file for each unit (i.e. header) being tested.
- Use the automatic registration facility, which requires the inclusion of `<boost/test/auto_unit_test.hpp>` and using the `BOOST_AUTO_TEST_CASE` macro to define each test case.
- For header-only usage, ensure that `BOOST_TEST_NO_LIB` is defined for compilation of all test files.
- Introduce a single `.cpp` file to hold the definition of `main`. This defines the empty macro `BOOST_AUTO_TEST_MAIN` and then includes `<boost/test/included/unit_test.hpp>`.

The use of automatic registration supports a fine-grained test-case style.

General Guidance

- Work as a team
  - Don't forget to swap partners — pairing is not marriage
  - Be clear about who is working on what class at any given point
- Write test cases before code
  - And ensure the code is sufficient for the task, and is neither clumsy nor overdesigned
- If there are problems, ask for help

For the first proper development sprint it makes a lot of sense to ensure that development skills are distributed as evenly as possible between pairs. If a team is made up of three rather than four people, the arrangement should switch around between a solo programmer and a pair.

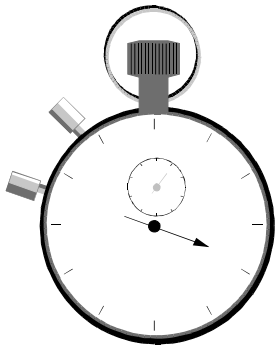
Suggested Activities for *Sprint 1*

- Functional...
  - High priority items
  - High risk items that involve the need for some experimentation
- Developmental...
  - Ensure that the partitioning of the code aligns with the development pairings as closely as possible

A suggestion is offered as to what to focus on for this sprint's increment.

Sprint 2: Construction

- Intent
  - Continue development, revising as necessary
- Content
  - General guidance
  - Suggested objectives



General Guidance

- Estimate how much work can be completed based on the last sprint
  - Did you do more or less than expected?
- Work as a team
  - Don't forget to swap partners
  - Be clear about who is working on what class at any given point
- If there are problems, ask for help

Begin the sprint by reviewing progress made in the previous sprint. Highlight what was successful in the previous sprint and what needs to be improved, so that, in terms of practices, you have an opportunity to continue doing what worked and react to any problems or shortfalls in expectation.

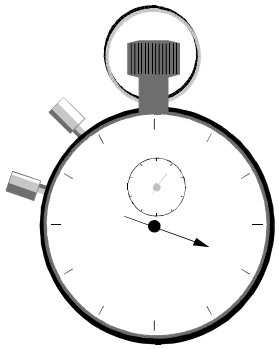
Suggested Activities for *Sprint 2*

- Functional...
  - Any outstanding items from previous sprint
  - High-priority items
- Developmental...
  - Ensure that I/O handling and calculation logic remain separate
  - Refactor duplicate or unused code

Complete anything that was only partly completed in the previous iteration and draw in other high or medium priority requirements. Ensure that the quality of both production and the test code is maintained, so that duplicate code, inconsistencies, unused code, etc, are all addressed.

Sprint 3: Completion

- Intent
  - Finish up!
- Content
  - General guidance
  - Suggested objectives
  - Outroduction



General Guidance

- Estimate how much work can be completed based on the last sprint
  - Did you do more or less than expected?
- Work as a team
  - Be clear about who is working on what class at any given point
  - Be clear about what tasks are outstanding to reach a reasonable state of completion
- If there are problems, ask for help

In the final sprint the emphasis is on completion. You want to make sure that you cover the minimum usable subset of functionality. You also want to ensure that requirements are completed with full scope or reduced scope. Completion is considered to be in terms of tests on all code. A state of partial completion is one to avoid since it indicates wasted work: effort invested in code that does not contribute to the overall, externally visible functionality.

Suggested Activities for *Sprint 3*

- Functional...
  - Any outstanding items from previous sprint
  - Any remaining high-priority items
  - Any lower-priority items that are easy to complete
- Developmental...
  - Tidy up production and test code
  - Refactor duplicate or unused code

If you are in a position to tackle lower priority items, select requirements based on complexity or, rather, simplicity — be sure that you can start and complete the requirements within the given time, so avoid anything that looks too challenging. Alternatively, if you have completed the minimum usable subset, you may also want to try alternative implementation strategies for your classes, refactoring the internal structure without breaking existing tests. Alternatively, focus on productisation aspects, such as documentation and quality of interface.

Outroduction

- Agile development processes aim to provide sustainable development in the face of change
  - The processes, and the code, travel light
  - Progress is visible and feedback is built in
  - Libraries offer developers a boost (sic)
- TDD applies these principles in the micro process at the minute-to-minute level
  - Higher confidence in quality of code
  - Higher confidence in ability to change code