the eclipse way
processes that adapt

Erich Gamma, John Wiegand
Eclipse Project Management Committee Members
IBM

outline

- on the way to 3.1
- the Eclipse way
on the way to 3.1...

3.1 themes

- scaling up
- enterprise ready
- design for extensibility: be a better platform
- simple to use
- rich client platform improvements
- J2SE 5 support
- broadening the community
JDT Quick Fixes

```java
public void conditionCheck(boolean a, boolean b) {
    if (a) {
        // Add else block
        // Invert condition
    } else {
        // Remove extra parentheses
        // Split line condition
    }
}
```

```java
if (element instanceof IdlTable) {
    // Introduce new local with cast
}
```

```java
String[] names = new String[] {"equa", "equals"};
for (int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}
```

JDT UI

```
Java Settings
Define the Java build settings.
```

```
Source
Projects
Libraries
Order and Export
```

```
Remove from Build Path
```

```
Configure Inclusion/Exclusion Filters
```

```
Do not include folders:
```

```
Exclude Source Folders:
```

```
Default output folder:
```

```
Browse...
```

The Eclipse Way | © 2005 by International Business Machines; made available under the EPL v1.0
JDT NLS support

```java
String s = Messages.getString("title");
String s = Messages.getString("externalized_value");
String s = Messages.getString("error");
```

Ant

```
<target name="init">
    <echo msg="JDK 1.5"/>
    <echo msg="classpath: test.jar"/>
</target>

<target name="default" depends="init" description="">
    <echo msg="JDK 1.5"/>
</target>
```

Java

**Description**

Compile a Java source file.
**SWT**

![SWT Image]

**PDE/RCP**

![PDE/RCP Diagram]
user assistance

outline

- on the way to 3.1
- the Eclipse way
why “the eclipse way”?

- our process has evolved over time
  - stay aware, adapt, change
- our process has helped us (the platform) to achieve:
  - predictability
  - quality delivery on time
- share what we have learned
  - influence “by example” – similar as JDT

—I've been following the development of Eclipse for some time and I'm continually amazed as the extended team hits each projected milestone and ship date with precision. Until this experience, I was convinced that such a feat was impossible for a software project of any size and complexity.

So... the question is how is this really accomplished? And the immediate follow-up of course is, how can this process be replicated elsewhere?

- a post to the eclipse developer’s mailing list
**eclipse practices**

- continuous testing
- end game
- always have a client
- component centric
- dynamic teams

**it's all about feedback!**

- continuous integration
- milestones first
- early incremental planning
- retrospectives

- consume your own output
- community involvement
- new & noteworthy

- continuous testing
- reduce stress
- explore

- built to last
- drive with open eyes

- validate
- update
- learn

- transparency
- show progress
- attract to latest

- validate
- update
- learn

- show progress
- attract to latest

- validate
- update
- learn

- transparency
- show progress
- attract to latest

**project rhythm**

- continuous testing
- end game
- always have a client
- component centric
- dynamic teams

- milestones first
- early incremental planning
- retrospectives

- consume your own output
- community involvement
- new & noteworthy

- continuous testing
- reduce stress
- explore

- validate
- update
- learn

- transparency
- show progress
- attract to latest

- validate
- update
- learn

- transparency
- show progress
- attract to latest

- validate
- update
- learn

- transparency
- show progress
- attract to latest

- validate
- update
- learn

- transparency
- show progress
- attract to latest
eclipse release cycles

- **milestones first**
  - break down release cycle into milestones
    - we currently use 6 weeks
  - each milestone is a miniature development cycle
    - plan, execute, test, retrospective
    - teams refer to the release plan when creating milestone plans
    - assign plan items to a milestone
    - milestone plans are public
  - result of a milestone
    - milestone builds: good enough to be used by the community
  - at the end of each milestone we do a retrospective
    - what went well, what did not?
  - milestones reduce stress!

- before/after
  - quality
  - ready to ship
continuous integration

- fully automated build process
- build quality verified by automatic unit tests
- staged builds
  - nightly builds
    - discover integration problems between components
  - (weekly) integration builds
    - all automatic unit tests must be successful
    - good enough for our own use
  - milestone builds
    - good enough for the community to use
- reality: build failures occur
  - rebuild to create acceptable integration, milestone builds.

always beta

- each build is a release candidate; we expect it to work
- results of the build process and the automatic tests
  - indicate where we are
- as tool makers we use our own tools
  - component team – use the latest code daily
  - project team – use integration builds (weekly)
  - community – use milestone builds
- continuously Consume Our Own Output
community involvement

- requires transparency
  - community needs to know what is going on to participate
- requires open participation
  - We value the contributions of the community
- we are the community
- problem: no one knew what was in a milestone,
  - so there was no incentive to move to milestone builds
  - so we received minimal feedback
    - more stale defect reports
    - quality suffered
- solution: publish new and noteworthy
  - advertise what we have been doing

testing

- continuous testing
  - end game
  - always have a client
  - component centric
  - dynamic teams
- API first
- early incremental planning
- retrospectives
- continuous integration
  - drive with open eyes
  - reduce stress
- consume your own output
  - enable
  - community involvement
  - attract to latest
  - new & noteworthy
  - validate
  - update
  - show progress
  - learn
  - built to last
  - enable
  - explore
testing

- innovate with confidence
- continuous incremental design
- > 20,000 JUnit tests
  - tightly integrated into the build process
    - tests run after each build (nightly, integration, milestone)
    - integration builds are only green when all tests pass
- test kinds
  - correctness tests
    - assert correct behavior
  - performance tests
    - assert no performance regressions
    - based on a database of previous test run measurements
  - resource tests, leak tests
    - assert no resource consumption regressions

---

test report

---

performance test report

Performance Results

finishing

end game

continuous testing

continuous integration

consume your own output

community involvement

enable

attract to latest

new & noteworthy

milestones first

show progress

update

learn

retrospectives

API first

early incremental planning

validate

update

learn

validate

always have a client

component centric

dynamic teams

built to last

show progress

update

learn

retrospectives

API first

early incremental planning

validate

update

learn

retrospectives

always have a client

component centric

dynamic teams

built to last

show progress

update

learn

retrospectives

API first

early incremental planning

validate

update

learn

retrospectives

always have a client

component centric

dynamic teams

built to last

show progress

update

learn

retrospectives

API first

early incremental planning

validate

update

learn

retrospectives

always have a client

component centric

dynamic teams

built to last

show progress

update

learn

retrospectives

API first

early incremental planning

validate

update

learn

retrospectives

always have a client

component centric

dynamic teams

built to last

show progress

update

learn

retrospectives
endgame

- convergence process applied before release
  - sequence of test-fix passes
    - community event
  - with each pass the costs for fixing are increased
    - higher burden to work on fix for a problem
    - higher burden to release a fix for a problem
    - focus on higher priority problems

```
# fixed bugs
447 608 301 85
eclipse 3.0 Release
May 21 May 28 June 11 June 20 June 25
time
```

endgame

- endgame endurance
  - we are only effective for so long
  - distribute Quality/Polish effort throughout the release
- shared responsibility and commitment
  - we all Sign off
[platform-releng-dev] PDE status for I-20040625-1208

- "An enthusiastic GO from PDE" - Cherie
- "The best build of the year!" - Dejan
- "An Eclipse build that the whole family can enjoy" - Wassim
- Your PDE team.

decompression

- recover from release
- retrospective of the last cycle
  - achievements
  - failures
  - process
  - cross-team collaboration
- start to plan the next release and cycle
where the time goes

- release cycle 12-16 months
  - milestones - 9 months
  - endgame – 1-2 months
  - decompression – 1 month

planning

- API first
- early incremental planning
- continuous integration
- continuous testing
- drive with open eyes
- validate
- reduce stress
- transparency
- show progress
- update
- community involvement
- attract to latest
- new & noteworthy
- retrospectives
- validate
- enable
- learn
- enable
- built to last
- dynamic teams
- component centric
- explore
- always have a client
early planning

- release themes establish big picture
  - community input
  - requirements council new source of input

- component teams define component plans

- PMC collates initial project plan draft
  - tradeoff: requirements vs. available resources
  - Committed, Proposed, Deferred

the plan is alive

- the project plan is updated quarterly to reflect
  - progress on items
  - new items
  - input from the community
  - becomes final at the end of the release

- before: static plans
  - accurate once, but no early feedback: non-existent until late in the cycle.
ongoing risk assessment

- address high risk items and items with many dependencies early
- maintain schedule by dropping items (if necessary)
  - we will drop proposed items
  - we hate to drop committed items
  - prefer fewer completed items than more items in progress
- high risk items are sandboxed to reduce risk to other items
  - prefer to serialize highest risk items (to minimize integration pain)

collective ownership

- PMC meets at least once a week
- all component leads and the PMC meet for a weekly planning call
  - status
  - planning
  - identification of cross-component issues
  - meeting notes posted to the developer mailing lists
- dynamic teams are established for solving cross-component issues
  - one cross-component issue per dynamic team
  - members are key developers from all effected components
  - find, implement, and roll-out solution of the assigned cross component issue
  - represented in the weekly planning calls
built to last

- deliver on time, every time
  - decisions in this release impact what we can do next release
  - must preserve architectural integrity

- deliver quality
  - innovation with continuity
  - need to have a solid foundation
    - scalable
    - performant
    - stable
how buildings “survive”

• Stewart Brand: how buildings learn
  – what happens after they’re built
  stuff: furniture
  services: electrical, plumbing (7-15y)
  structure: foundation, load bearing walls (30-300y)
  site: geographical setting (forever)

• layers:
  • evolve at different rates during the life of a building
  • shear against each other as they change at different rates
  • an adaptive building must allow slippage
    ➢ a building that lasts is adaptive and can change over time

structure foundation

• the eclipse plug-in architecture
• everything is a plug-in
  • simple and consistent
services plumbing: APIs

- APIs matter
  - define consistent, concise API
  - don’t expose the implementation
  - develop implementation and client at the same time
- define APIs for stability
  - binary compatibility is highest priority
  - we would rather provide less API than desired (and augment) than provide the wrong (or unnecessary) API and need to support it indefinitely

component centric

- component centric development
  - a team is responsible for one or more component
- dependencies through APIs
  - ensures high velocity development inside a component
  - *eclipse 3.1* provides tools support to check for API access violations
  - define producer/consumer relationships among components
    - tension among components is healthy for coming up with good component interfaces/APIs
APIs first

- APIs don’t just happen; we need to design them
- specifications with precisely defined behavior
  - what you can assume (and what you cannot)
  - it works ≠ API compliant
- must have at least one client involved, preferably more
- need an API advocate
  - we all care about having sustainable APIs
  - need someone who lives and breathes APIs

practical extensibility

- extensible in ways that are known useful
  - needed by us, requested by others
- we don’t provide hypothetical generality - we want to be extensible in ways that matter
  - don’t over generalize
API tension

- conflict
  - APIs need to be implemented and used to be right
    - requires iteration
    - needs external client
  - stable API necessary
    - for widespread adoption

- resolution
  - don’t commit API before its time
    - APIs can (and should) change during the release to accommodate new requirements, experience

summary

- continuous testing
- continuous integration
- milestones
- API first
- early incremental planning
- built to last
- component centric
- dynamic teams
- always have a client
- end game
- reduce stress
- drive with open eyes
- validate
- community involvement
- consume your own output
- enable
- attract to latest
- new & noteworthy
- transparency
- show progress
- update
- learn
- retrospectives
- validate
confessions

- ripples - a change in a lower layer that propagates up
  - we often underestimate the impact on upper layers!
  - consequence: end game swirls and stress
- ground rule:
  - you are not done unless the upper most layer has digested the ripple
- dynamic teams – not all efforts are successful
  - require leadership, feeling of responsibility
  - require even more detailed planning
- we drop features!
  - but we hate to drop committed items

summary: how extreme is eclipse?

- testing early, often and automated
- incremental design
  - Yes but API stability is important to us
- daily deployment
- customer involvement
  - even better we have an active community
- continuous integration
  - nightly, weekly, milestone
- short development cycles
  - 6 week cycles
- incremental planning
  - start with embryonic plan
  - refine after each milestone
conclusion

- the team makes the process work
- the team defines and evolves the process