Contributing Code to Eclipse: Why To. How To.

Bjorn Freeman-Benson
Technical Director, Eclipse Foundation
August 31, 2005
Presented at Eclipse World 2005, New York, NY

Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

- This is a talking-to-you Q&A presentation – there are no demos or hands-on exercises, but you are encouraged to ask questions, provide clarification, etc.

Why Contribute?

- Eclipse is about many things, but one of those things is the Eclipse ecosystem and the pursuit of profit.
- Eclipse is not about giving away free software.
- Eclipse members are building products on top of the extensible frameworks – that is the value for everyone.

- You can use Eclipse without contributing.
  - No influence
- You can join an Eclipse project
  - Influence that project
- You can join an Eclipse project and the Foundation
  - Influence policy and direction

Why Contribute?

The main reasons:

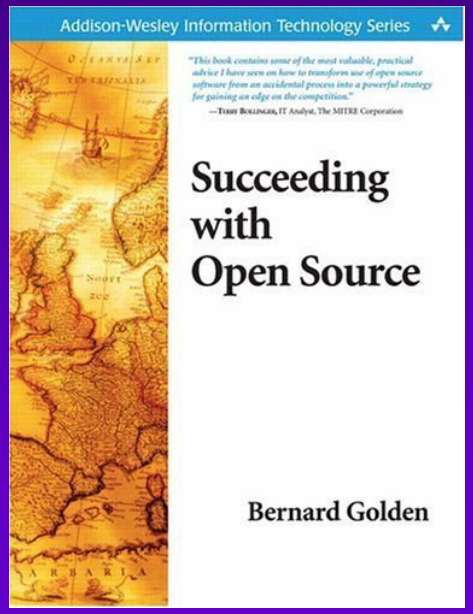
- Product dependency – you want to help direct the project in ways that will help your commercial product.
- Branding – you want to associate your company or product with the Eclipse brand and buzz.
- Commoditization – staying aware of the trends

Other reasons:

- Altruism / Golden Rule
- Developer morale
- Decreased maintenance
- Increased quality (from community process)

Open Source Maturity

- Adopting, contributing to, or joining an open source project requires consideration.
- Eclipse projects vary in maturity, but the top-level projects are serious software built by talented professional.
- Eclipse projects evaluate very favorably under the Open Source Maturity Model.
  - Well defined processes
  - High quality releases
  - Predictable, honest schedules
  - Extensible frameworks
  - Exemplary tools



Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

Open Source Process

- The essence of the Open Source development model is that combining the availability of source code with an open, transparent, collaborative process enables the rapid creation of solutions.
  - SourceForge.net document A01
- The Open Source community argues that the collaboration between developers and users promotes a higher standard of quality and helps to ensure the long-term viability of the project.

- Mostly, Open Source development is just like general software development. With two exceptions[1], people who are good at general software development will also be good at open source development.

[1] "Foreshadowing: the sign of quality literature" (for the exceptions see slide 15)

Meritocracy

- Open Source projects are run as *meritocracies*.
- Meritocracy is a system of government based on rule by ability (merit) rather than by wealth or social position; “merit” means roughly intelligence plus effort.
- In other words, *trust earned through competence*. It is not possible to buy influence in an Open Source project.
  - Note that this is the same social process that exists, more or less, in all corporate environments. When someone joins your team, you analyze their merits and decide how much to trust them – more or less than their official position.
  - Example: Microsoft. Bill Gates is the CTO, with CEO influence.
  - Example: Amazon.com. One executive had never shipped an Amazon product and thus was less influential than he would, or should, have been.

Open and Transparent Process

- "Open Source" is about more than just the source code:
  1. Free redistribution
  2. Accessible and available source code
  3. Allow modifications and derived works
  4. No discrimination
  5. …and a few other clauses
  - The Open Source Definition, opensource.org

- But Eclipse Open Source is even more than that. Eclipse uses the model quoted previously from SourceForge.net – that Open Source is about an *open and transparent process*.

Open, Inviting, Transparent & Collaborative

- Open – the development process allows contributions from outside the core team; “open” is the inward flow of information into the project
- Inviting – the development process encourages contributions from outside the core team
- Transparent – the code and the development process are available for inspection; “transparent” is the outward flow of information from the project
- Collaborative – the project is a collaborative effort between the users and the developers
  - The developers write code; the users use it.
  - The users request features; the developers listen.

Roles in the Meritocracy

- The lowest role is "silent user" – one who downloads and uses the project, but other than increasing the download stats, does not provide content.
- Next is "early adopting user" – an important testing role
- "Patch contributor" is someone who does occasional development and submits the changes via patch files
- "Developer" is an intermediate step
- "Committers" actively and regularly work on the project
- The position of highest respect is "project lead".
  - Linux has "integrator" instead of "lead" (see next slide)

Project Management Style

- The standard mythology around open source is that the projects are undirected and staffed by part-time volunteers.
- That may be true for some open source projects (hey, it might be true for some commercial projects!), but it is not a prerequisite of Open Source.

- Most large and successful open source projects (such as Eclipse) are staffed by paid developers and managed by professionals.
- Linux is a partial exception: it uses the un-directed/semi-directed bazaar style.

Joining an Open Source Project

- Joining a open source (OS) project is like joining a social club – you must convince the existing members of your value. There are three basic strategies for gaining the reputation necessary to join a project:
  - New: Join a low velocity project or start a new open source project. As the project grows in reliability, predictability, and results, your reputation in the community increases.
  - Full-time: Your employer commits you to an open source project on a full-time, or nearly full-time, basis - working full-time on the project allows you to quickly gain the respect of your peers and become a committer.
  - Part-time: You contribute on a part-time basis, working on a particular aspect of a project. Because the high velocity projects have full-time Committers, it is hard for a part-time developer to keep up. Part-time contributions are successful in corners of the space such as writing tutorials and articles, updating the website, coding non-critical-path components, etc.`

Communication

- One of the big differences between an OS project and a normal internal company CS project is communication.
- The OS developers tend to be more distributed than CS developers.
- OS projects often span multiple companies.
- OS projects are more directly connected to their customers through mailing lists and defect tracking, whereas CS projects have a “support organization” and even a marketing team as filters.

- The net net is that OS projects and developers have to be better at communication.

The Two Exceptions

- The general software developers who are not good at open source are those who are not good at communication and consensus building.
- Communication – developers who “go dark” do not do well in OSS. Developers have to be actively talking to the users and other developers.
- Consensus – developers who “command” do not do well in OSS. The fact that open source projects are not hierarchical is very difficult people who are used to control. The users do not work for the same company. Often the developers do not all work for the same company. Leadership is by merit and consensus rather than by dictate.

Modularity

- Successful open source software requires a high degree of modularity to support the distributed development process.
- Well defined APIs are essential because of two *network effect* characteristics of open source software:
  - Successful projects have a very quick adoption rate
  - Changing too much, too often causes a very fast rejection
  - *Bad news travels quickly*
- At Eclipse, the mantra is “API First”
- A public method is not an API
  - (more about this on slide 60)

Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

Eclipse

- The purpose of the Eclipse projects is "a vendor-neutral, open development platform supplying frameworks and exemplary, extensible tools... [the] tools are extensible in that their functionality is accessible via documented programmatic interfaces."
  - Eclipse Foundation Bylaws

- Essential to that purpose is the development of three inter-related communities around each project: committers, plug-in developers, and users.
- The absence of any one or more of these communities as proof that the project is not sufficiently open, transparent and inviting, and/or that it has emphasized tools at the expense of extensible frameworks or vice versa.

Open Source Communities

- Typical open source projects have two communities: committers and users.
  - Some projects have one community because the committers are the only users.
  - Some projects have three communities: committers, non-committer contributors, and users.
- Eclipse projects have three communities:
  - Committers and contributors
  - Users – those who use the exemplary tools
  - Plug-in developers – those who use the extensible frameworks

Committers

- An active, open, transparent, and inclusive community of Committers, developers, and other non-Committer contributors is essential for the health of the project.
- Attracting new contributors and committers to an open source project is time consuming and requires active recruiting, not just passive "openness".
  - The project leadership must go out of its way to encourage and nurture new contributors.
- A Committer community comprised entirely, or even in the majority, from a single company is contra-indicated for a project's long-term success as an open source project.
- The long term viability of an open source project is a function of its developer community, not of the code base.

Users

- An active and engaged user community is proof-positive that the project's exemplary tools are useful and needed.
- Furthermore, a large user community is one of the key factors in creating a viable ecosystem around an Eclipse project, thus bringing additional open source and commercial organizations on board.
- Like all good things, a user community takes time and effort to bring to fruition, but once established is nicely self-sustaining.

Plug-in Developers

- An active and engaged plug-in developer community is only way to prove that an Eclipse project is providing extensible frameworks and extensible tools accessible via documented APIs.
- Reuse of the frameworks within the companies that are contributing to the project is necessary, but not sufficient to demonstrate a plug-in community.
- Creating, encouraging, and nurturing a plug-in developer community outside of the project's developers takes time, energy, and creativity by the project leadership, but is essential to the project's long-term open source success.

Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

Background

- In the beginning, there was just one project: the IDE.
  In the beginning, the Eclipse Development Process (EDP) referred to the process the IBM/OTI team used to build the IDE.
  - (This is the process now known as “The Eclipse Way” (TEW).)
- Next, the Eclipse Consortium was created and a few dozen additional projects were created. The IDE was years ahead of the other projects and “EDP” still meant “TEW”.
- In 2004, the Eclipse Foundation was formed with Bylaws and an official Eclipse Development Process. Many new projects were started using a variety of individual processes. Slowly EDP is becoming the meta-process and TEW the Platform Project process.
  - Note that many Eclipse projects emulate TEW.

Guiding Principles (Quality)

- Quality - In this context, quality means extensible frameworks and exemplary tools developed in an open, inclusive, and predictable process involving the entire community.
  - From the "consumption perspective," Eclipse Quality means good for users (exemplary tools - cool/compelling to use, indicative of what is possible) and ready for plug-in developers (deliver usable building blocks - with APIs).
  - From the "creation perspective," Eclipse Quality means working with a transparent and open process, open (and welcoming) to participation from technical leaders, regardless of affiliation.
  - From the "community perspective," Eclipse Quality is that the community perceives quality, i.e., if the frameworks and tools are good enough to be used, then they have sufficient quality.

Guiding Principles (Reputation)

- Collective Reputation - Having the Eclipse name on a project provides a certain "goodness" to the project. And having great and amazing projects under the Eclipse banner provides a certain "goodness" to Eclipse. Correspondingly, having a highly-visible poor and/or failing project under the Eclipse banner detracts from that reputation.
  - A certain number of failures are expected in any research and development effort, thus we do not let the fear of failure prevent us from accepting interesting project. However, it is in the community's best interest to have a well-defined processes for identifying and dealing with failures when they occur.
- Meritocracy - Eclipse is a meritocracy.
- Evolving - the frameworks, tools, projects, processes, community, and even the definition of Quality continues to, and will continue to, evolve. Creating rules or processes that force a static snapshot of any of these is detrimental to the health, growth, and ecosystem impact of Eclipse.

Guiding Principles (Just Enough)

- Just Enough Process - The Eclipse Development Process should be "just enough" to ensure that the community's goals (quality, openness, etc), but no more - we want to make it easy and inviting for high-quality teams to build interesting tools at Eclipse. The entry bar should be "just high enough" to keep Eclipse great, but no more - we want to make it easy to experiment and explore new ideas while simultaneously supporting the ecosystem with strong releases. The processes and goals should make projects:
  1. Easy to propose
  2. Fairly easy to create
  3. Kinda hard to validate (e.g., exit incubation)
  4. Pretty tough to ship
  - The processes are designed to enhance the middle ground of continued quality growth in Eclipse projects by eliminating the two undesirable endpoints:
    - no entry bar results in a wild mish-mash of projects, and
    - an entry bar so high that nothing new ever gets started

Guiding Principles (Culture)

- Culture of Quality - The Eclipse projects are managed by different people and different companies, most already experienced in software engineering. We want to ensure that we share the best practices of all our experts so that all projects benefit.
- Eclipse Ecosystem - The Eclipse open source projects are distinct from the Eclipse membership in spite of the majority of the resources on the projects being donated by the members. The projects are managed for the benefit of both the open source community and the ecosystem members; these groups will, at times, have different goals.
  - Eclipse benefits when their interests align
  - Eclipse benefits when their interests provide cognitive dissonance that provokes creativity
  - Eclipse suffers when one side of this duality takes precedence

Eclipse Projects

- Officially, there are two kinds of Eclipse projects:
  - Top-level projects, a.k.a. Projects, a.k.a. PMCs.
    - Officially, PMCs are just collections of projects and do not have code of their own.
  - Sub-projects, a.k.a. Projects.
    - Sub-projects exist under the umbrella of PMCs.
- Originally, there was a three level hierarchy:
  - Platform – the base; solid and reliable
  - Tools – on top of the base; high quality
  - Technology – research and incubation
- Eclipse is moving towards focus-centric top-level projects (web, data, embedded, modeling, …) each containing multiple sub-projects at various levels of maturity.

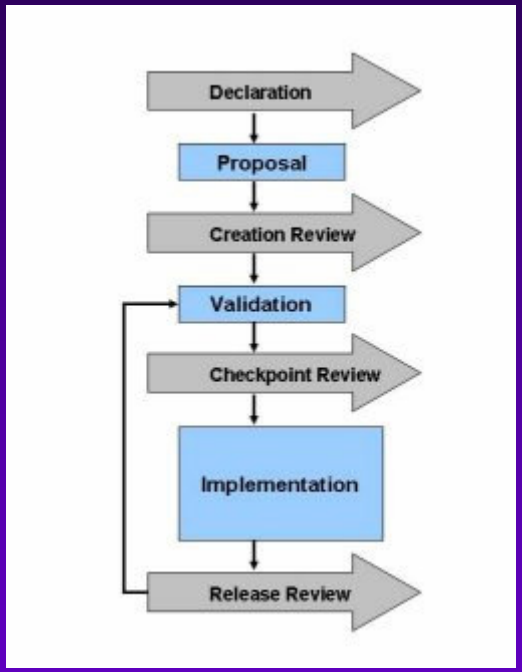
Top-Level Projects

- Eclipse top-level projects (i.e., PMCs) are about technical leadership in their specific area.
  - Leadership includes helping to create and define the Eclipse direction by participating in the Councils and creating the Eclipse Roadmap.
  - Leadership includes outreach activities and supportive marketing activities.
  - The consequence of the leadership required is that a top-level PMC is more than just a project management committee - it's almost a small startup in itself.
- Typically led by an Eclipse Strategic Developer, a top-level project is a focused collection of sub-projects.
  - Gather and create an active and diverse community of contributors. These communities do not just happen, thus a top-level PMC must actively recruit additional companies and individuals.

Project Lifecycle

- Pre-proposal
- Proposal
- Validation/Incubation – establish a fully-functioning open-source project
- Implementation/Mature
- Archived – after reaching the end of their natural lifecycle

- Official Reviews between each phase



Pre-Proposal Phase

- Clear and concise description understandable by the diverse Eclipse community and not just by specialists in the subject matter.
- Committed resources. Eclipse is not a dumping place for unwanted code. Projects involve substantial on-going development activity.
- Tools versus Run-times.
- Consistent with the Purposes as defined in the Bylaws.
- Vendor neutral and operating system agnostic.

Proposal Phase

- The proposers, the destination PMC and the community-at-large, collaborate in public to enhance, refine, and clarify the proposal.
- Approximately two months for simpler projects and longer for complex or controversial projects.
- Ends with a Creation Review.
  - Maturity plan?
  - Communities?
  - Collaborations?
  - Extensible frameworks?
  - Place in the Eclipse architecture?

Legal Paperwork

- Committers
  - Committer Questionnaire – information for the database
  - Committer Agreement – individual or member company
  - Individual Committer Employer Consent Form
- Contributions
  - Contribution Questionnaire is required for “significant” contributions
  - Approval required before code is committed to CVS
  - Contributions can come from anywhere (email, newsgroups, bugs, …), but should all be funneled through Bugzilla for traceability
  - IP Logs must be maintained
  - Many issues: appropriate, necessary, cryptography,license, …

Incubation Phase

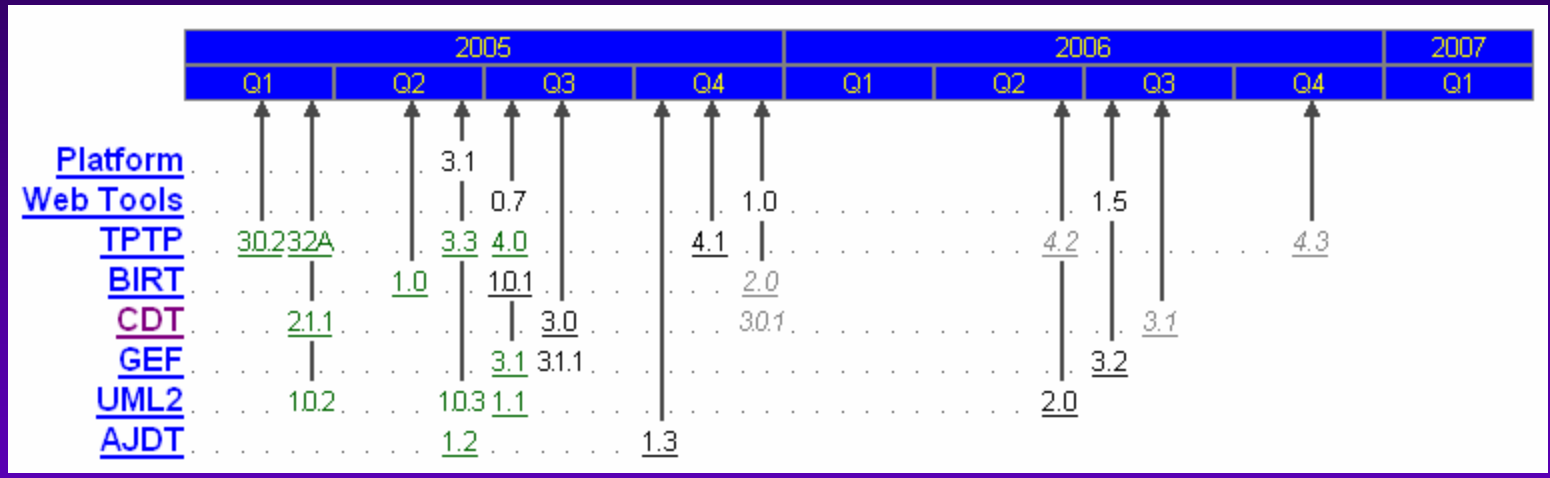
- About developing the process and the community.
  - While it might seem easy, history has shown that it takes experience to run an open, transparent, welcoming, and predictable software development process.
  - And history has shown that it takes a significant investment of time and energy to nurture a community of tool users and framework users around a new project.
- Includes typical software project management issues such as identifying critical use cases, producing a high level design, acquiring the necessary rights to all requirement intellectual property, and so on.
- Also includes creating viable user, plug-in developer, and committer communities around the project.

Checkpoint Review

- A working and demonstrable code base.
- Active communities
- Operating fully in the open using open source rules of engagement, e.g., open and transparent Bugzilla, project schedules, developing APIs, project decisions being made in public, etc.
- Team members have learned the ropes and logistics of being an Eclipse project. They “get it”.
  - Conforming to the EDP. Following the IP Policy. Participating in the larger Eclipse community.
  - The project is a credit to Eclipse and is functioning well within the Eclipse community.
- In-depth review of the technical architecture of the project, and its dependencies and interactions with other projects.

Mature Phase

- Status reports using the in-CVS XML file.



- IP Policy and Project Log.
- Active progress and regular Milestones.
- Three Communities (including proper voting)
- Branding

Schedules

- Strong emphasis on predictable schedule.
- Even stronger emphasis on an honest schedule.
- Especially critical for Eclipse projects due to external dependencies on the extensible frameworks.
- Best scheduling algorithm is under-promise and over-deliver.
  - Very important in open-source to allow enough time for community feedback that will, in turn, result in a really great framework and outstanding exemplary tools.
- The Eclipse “release train” has chosen the “TEW standard” six week milestones.

Release Review

- A time for boasting. A time for certification.

- APIs
- IP Policy conformance
- Architectural issues
- End of life
- Three communities
- Non-code aspects

APIs

- Eclipse Quality APIs have:
- Specification – a description of the cover story and the necessary details. Not just Javadoc. Should include what areas are likely to change in the future.
- Test suite.
- Implementation.
- One or more clients – best if written by a separate team.
- Support promise – implicit or explicit statement about the release-to-release stability of the API.

- "Provisional APIs" allowed for feedback prior to hardening.

Many More Details

- This talk just covers the highlights of the Eclipse Development Process.
- There are many more details
  - For example, the semantic meaning of version numbers (M.N.P).
  - For example, marking stale CVS modules.
  - For example, the project naming guidelines.
- Refer to the online documentation.

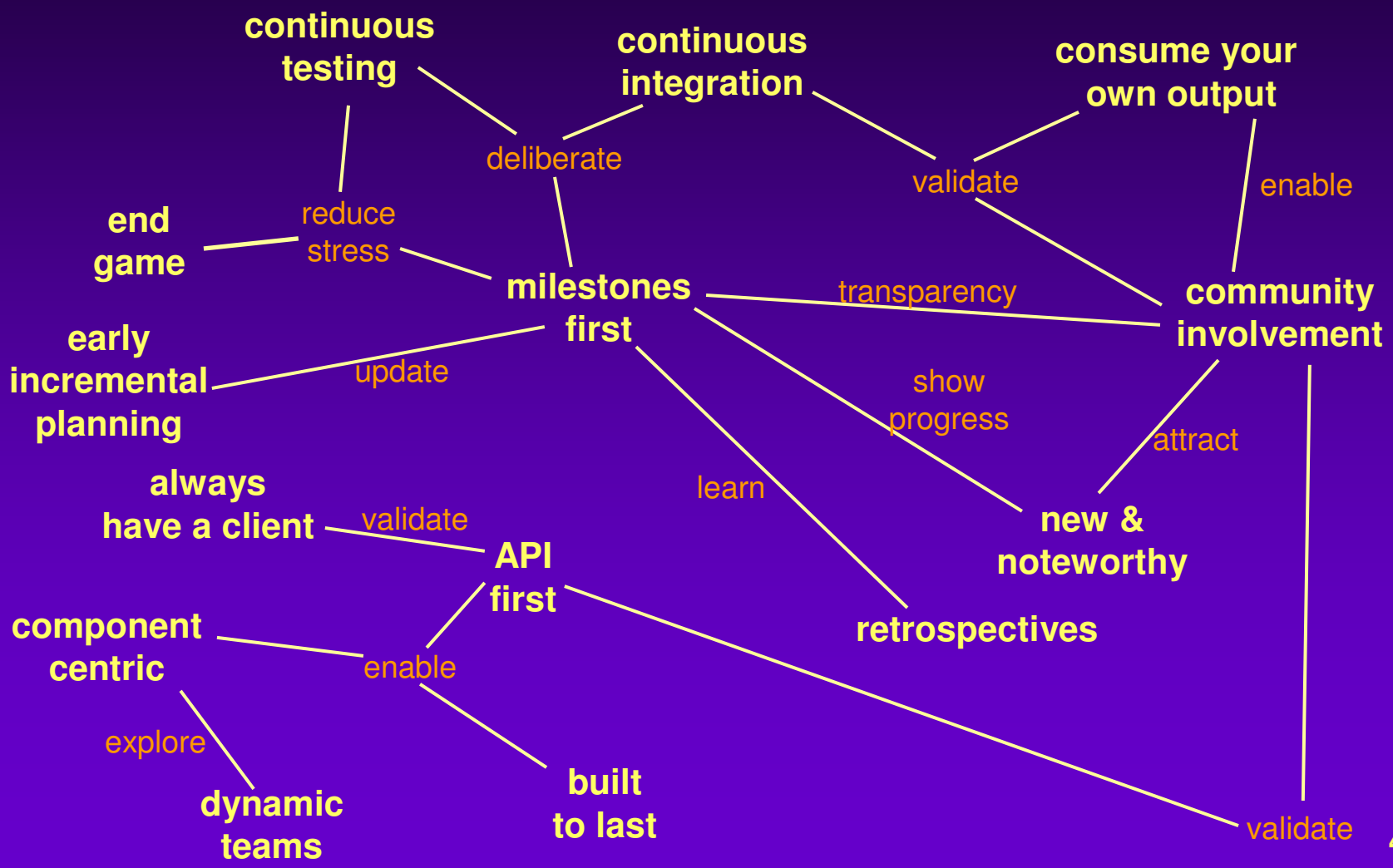
Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

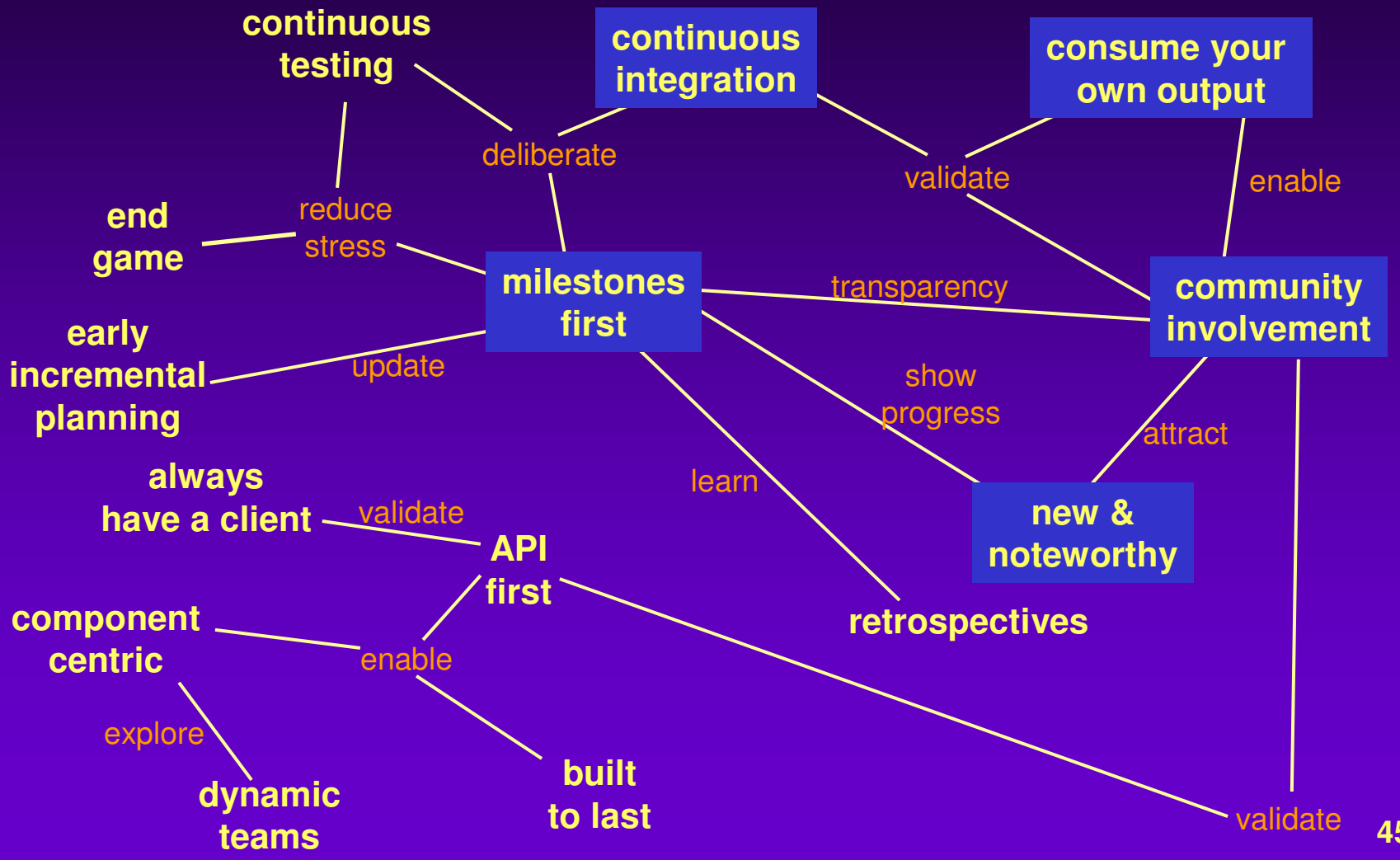
The Eclipse Way

- This material based on “The Eclipse Way” by John Wiegand and Erich Gamma, EclipseCon 2005
- “The Eclipse Way” (TEW) is the development process followed by the Eclipse Platform and JDT teams, and emulated by other Eclipse teams.
- TEW is one of many possible processes; TEW works well for this team; other Eclipse projects use minor or major variations based on what is best for that team.
- TEW is a (mostly) Agile process.
  - ***Individuals and interactions*** over *processes and tools*.
  - ***Working software*** over *comprehensive documentation*.
  - ***Customer collaboration*** over *contract negotiation*.
  - ***Responding to change*** over *following a plan*.

Platform Practices



Project Rhythm



Milestones First

- Yearly releases
  - 9 months of milestones
  - 2 months of end game
  - 1 month of decompression
- Six week milestones
  - Plans are public
- Each milestone is a miniature development cycle
  - Plan, execute, test, retrospective
- Milestones builds are good enough to be used by the community

Continuous Integration

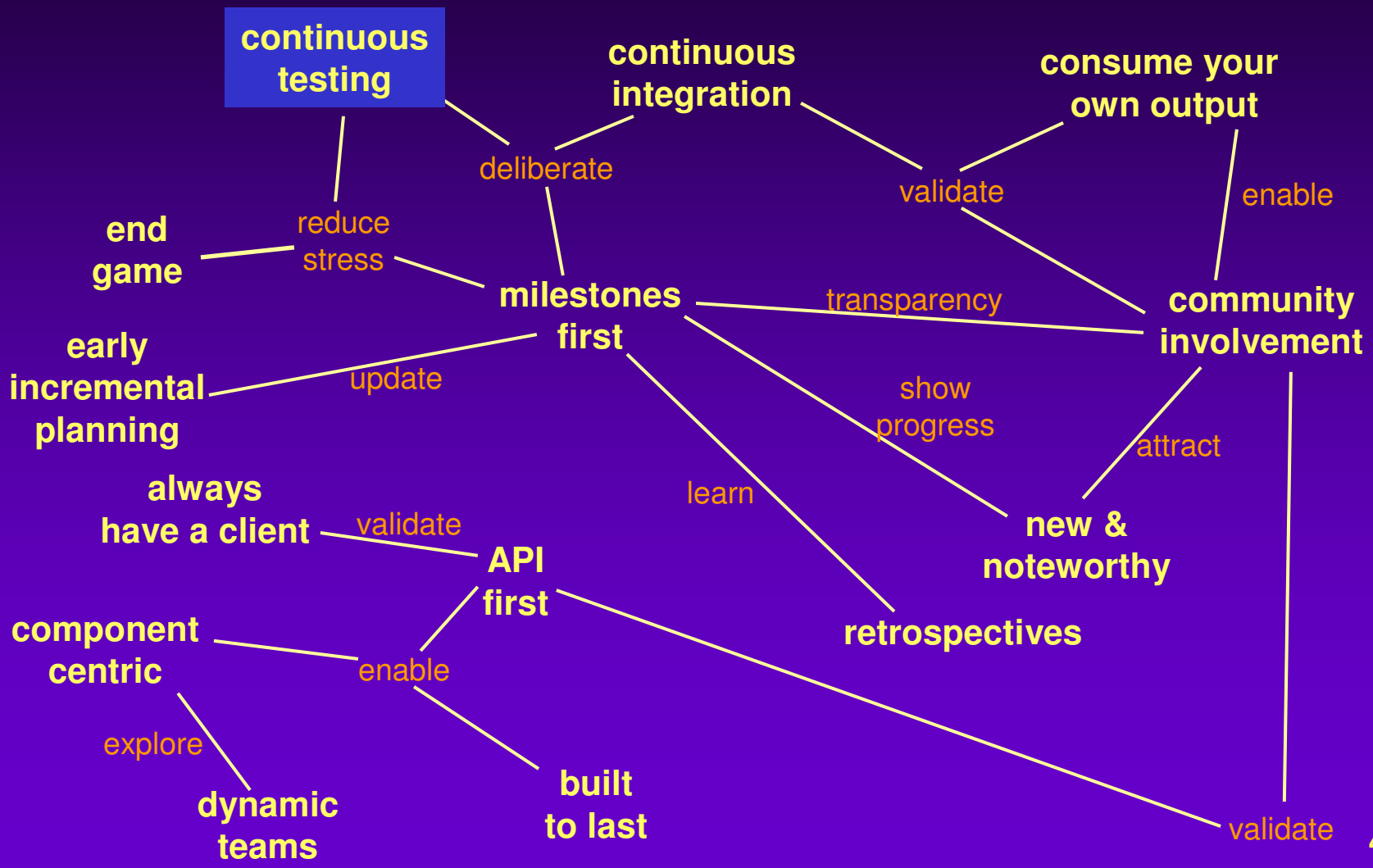
- Fully automated build and test process
- Nightly builds – discover integration problems between components
- Weekly integration builds – good enough for our internal use; all automatic tests must pass
- Milestone builds – good enough for community use

Community Involvement

- Always beta; each build is a release candidate
- Continuously use own tools as second line of testing

- Open and transparent is a lot of work
- The community needs to know where the project is
  - Example: New & Noteworthy
- Value the contributions of the community
  - Example: be careful what you say when you close a bug

Testing

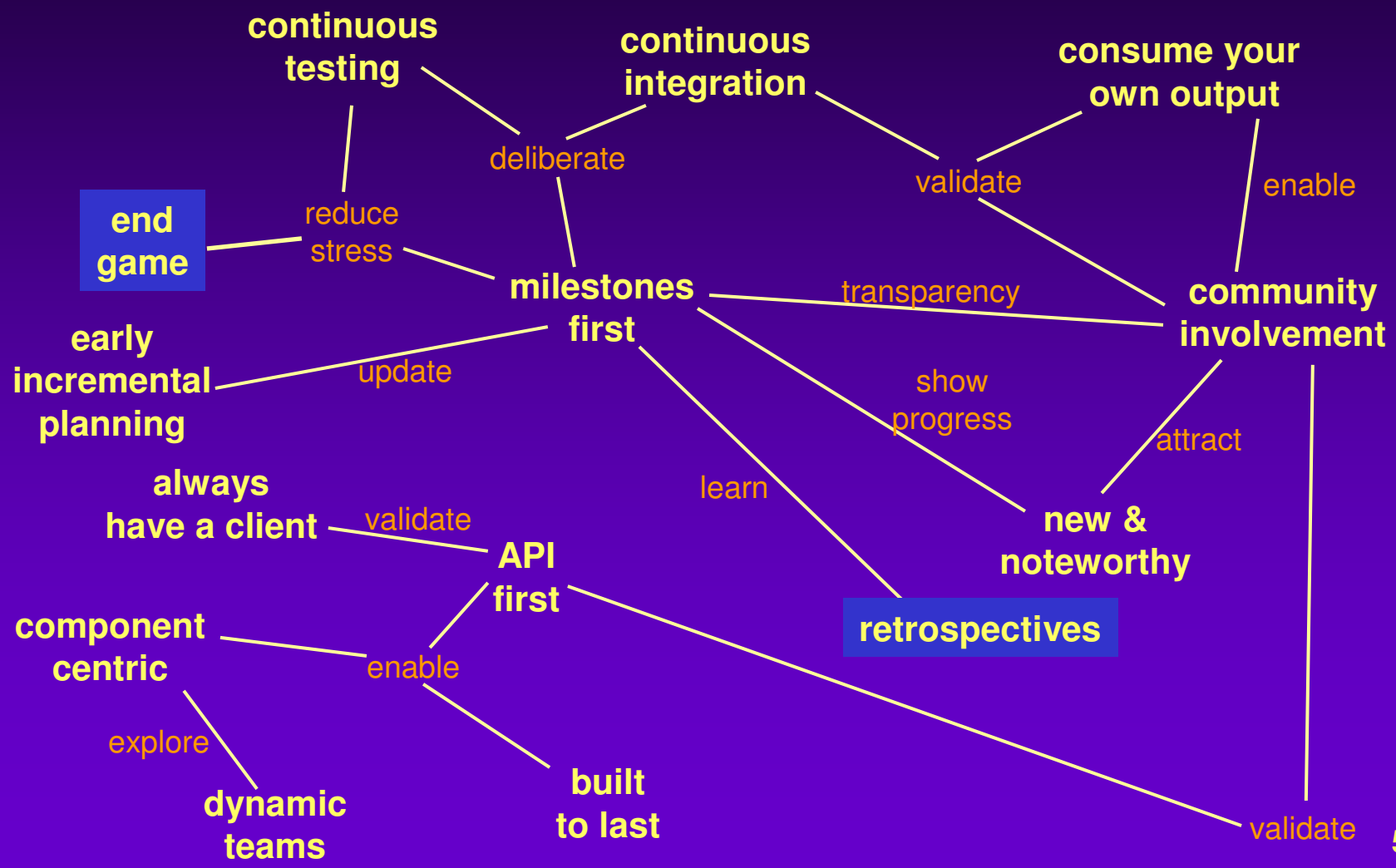

continuous testing
continuous integration
consume your own output
deliberate
validate
enable
end game
reduce stress
milestones first
transparency
community involvement
early incremental planning
update
show progress
attract
always have a client
validate
API first
learn
new & noteworthy
component centric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

Testing

- Motivated by Test Drive Development
- Innovate with confidence

- Correctness tests – assert correct behavior
- Performance tests – assert no performance regression
- Resource tests – assert no resource consumption regression

Finishing


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

End Game

- Convergence process applied before release
- Sequence of test-fix passes involving the entire community
- With each pass, the bar for fixing is increased
  - Focus on higher priority problems
  - Higher criteria to release a fix
  - At the end, it requires entire PMC approval to release a fix
- Each Release Candidate (RC) cycle is shorter than the previous one.

- The end-game can also be seen as the progressive removal of developers from the project. A “ramp down” that is necessary for stabilization and release.

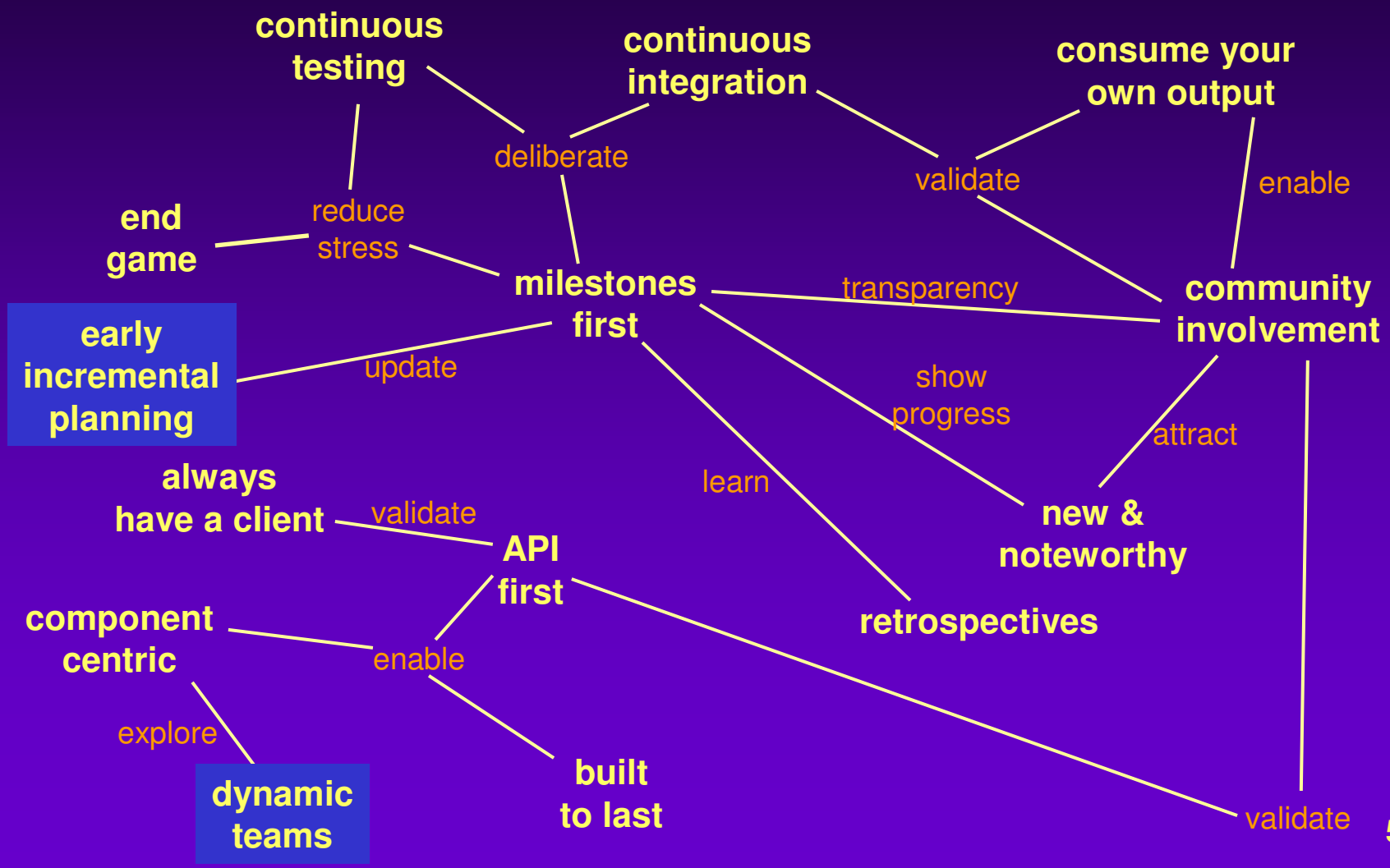
End Game (2)

- Why such a short (8 week) end game?
  - People are only effective at this kind of work for so long. After that, the Death March syndrome kicks in and hope is lost.
- How can this work?
  - Distribute the Quality & Polish items throughout the release.
  - Do not use a “feature complete” / “code complete” division where a feature is considered complete even if it has bugs.
    - That is bad Code Debt of unknown size.
- Keep the clients involved all along
  - If they only become involved at the very last minute, then the end game becomes very stressful and less likely to succeed.

Decompression

- Unofficially: July is **the** month for vacations in Ottawa ☺
- Recover from release
- Retrospective of the last cycle.
  - What went well?
  - What went poorly?
  - What can we change to avoid the flaws next time?
  - How are we doing on cross-team and cross-project collaboration?
- Start the next release planning.
  - There are always too many requests…

Planning


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

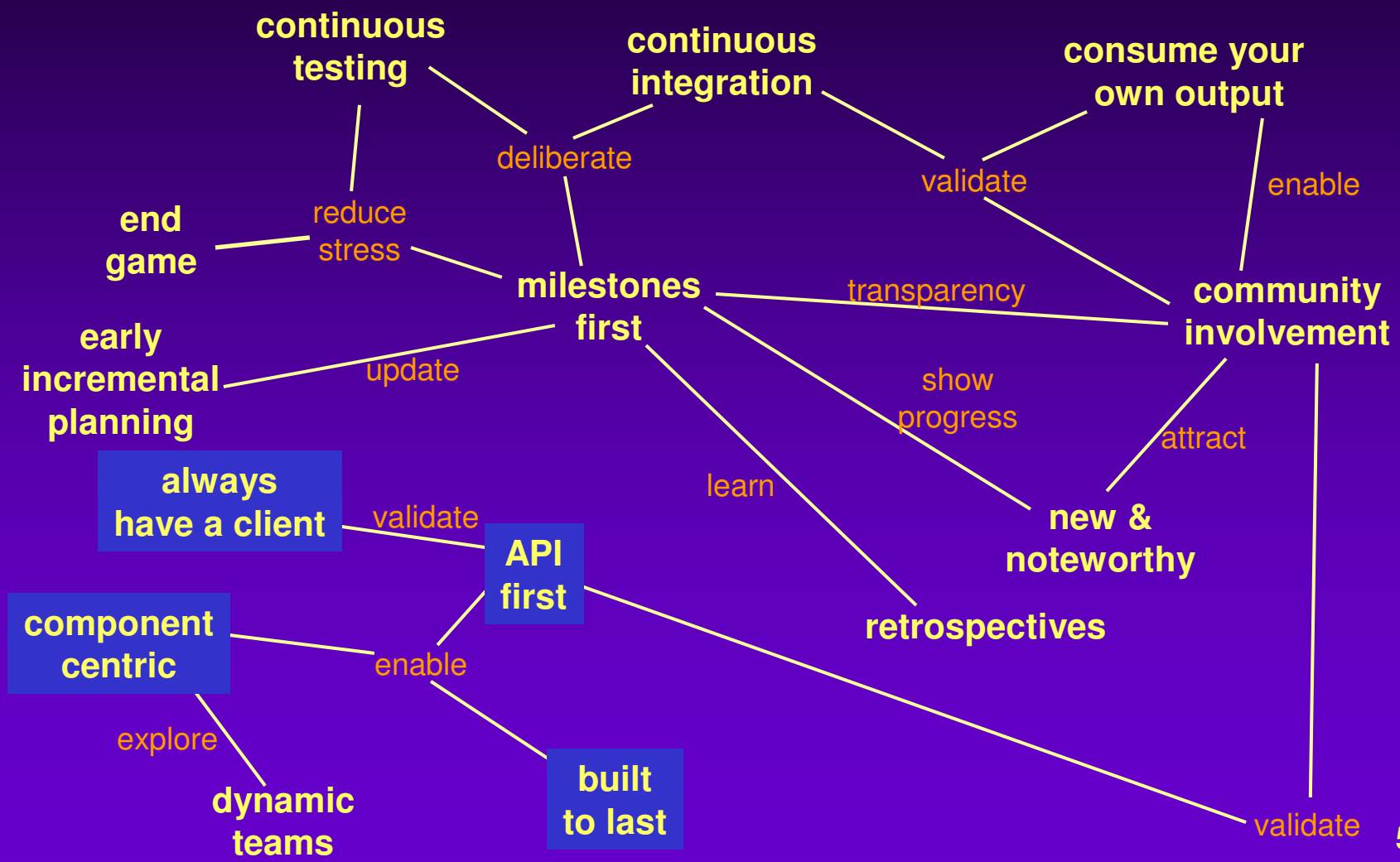
Early Planning

- Themes and Priorities to establish the big picture
  - Community input (as with everything)
- Teams define their own plans
- PMC collates into overall plan
  - Requirements versus available resources
  - Priorities: Committed, Proposed, Deferred
- Plan is flexible; changes each quarter to reflect
  - progress
  - new items
  - input from community

Ongoing Risk Assessment

- Address high risk items and items with many dependencies early.
- Maintain schedule by triage
  - Best to under-promise (committed) and over-deliver (proposed and deferred).
  - Worst to over-promise and under-deliver; fight against this tendency as everyone always wants to promise the moon
- High risk items are sandboxed to reduce risk to other items
  - Prefer to serial high risk items to reduce integration pain
  - True story: at least one feature has been tried three times by the Platform team with three failures; each time it has been discarded rather than slipping the schedule

Built To Last



Built To Last

- Deliver on time, every time
- But decisions in this release will impact all future releases
  - Preserve architectural integrity
- 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
  - Prefer less API and augment later, than provide wrong or unnecessary API and have to support it indefinitely

APIs First

- APIs don't just happen; they must be designed
- Specifications with precisely defined behavior
  - What you can assume (and what you cannot)
  - Just because it works doesn't mean it is API
- Must have at least one client involved, preferably more
- All clients, including all internal components, must be able to implement all their behavior using APIs only.
- Each project needs an API advocate.
  - A senior technical leader
  - Who lives and breathes APIs
  - Otherwise it is too easy to die the death of a thousand cuts

API Tension

- Extensible in ways that are known to be useful
- Do not provide hypothetical generality
  - It doesn't matter yet and you'll be wrong anyway
  - Don't over generalize

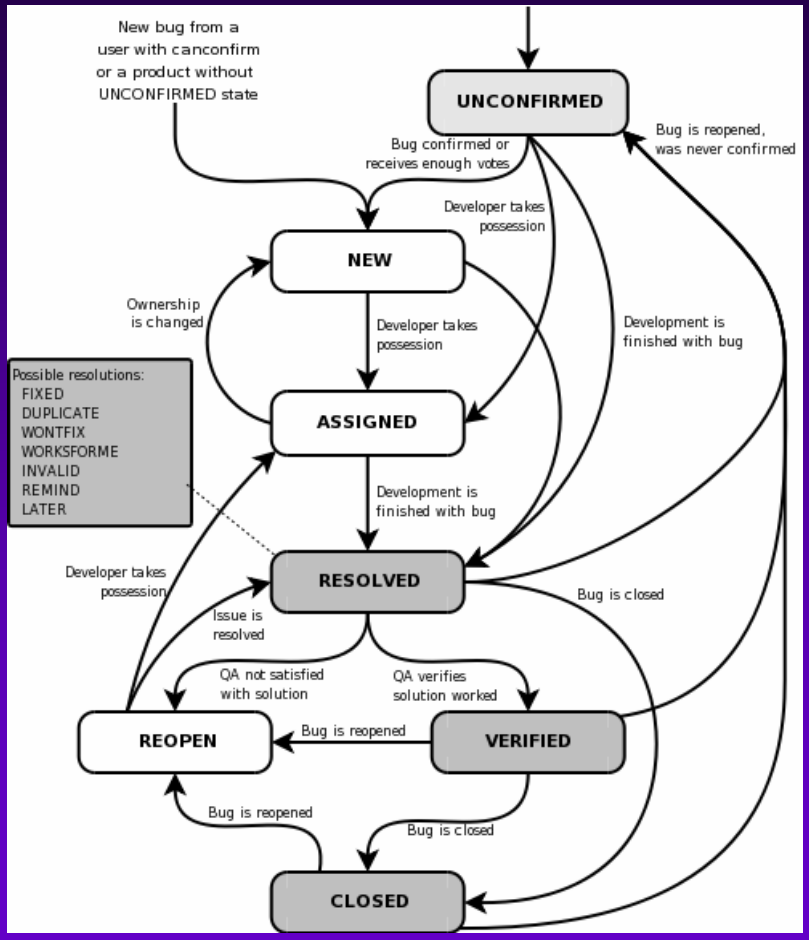
- Conflict between needing to iterate an API to get it right and needing a stable API for widespread community adoption
- Resolution: "no API before its time"
  - APIs will change during the release to accommodate new requirements and experience
  - APIs will be frozen at the release

API Issues

- Definition
  - Do the internal escape through the API?
  - What about meta-data? file formats? XML schemas? generated artifacts (e.g., generated code)?
- An API tells a story
  - Is the story that the API tells concise and precise or vague and wandering? Is the story clear and understandable?
- Good API
  - Doesn't have unnecessary bits.
  - Discusses thread safety and event ordering.
  - APIs versus SPIs
  - Where the API will grow in the future
  - Etc.

Bugzilla Process

- Users own: component, version, platform, OS, severity, summary, description.
- Committers own: status, resolution, priority.
- Daily triage and assignment.
- Everything is tracked: defects, enhancements, patches, …

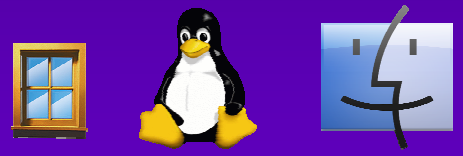


Outline

1. Why Contribute?
2. Open Source Development Process
3. The Three Eclipse Communities
4. Eclipse Development Process
5. “The Eclipse Way”
6. Now What?

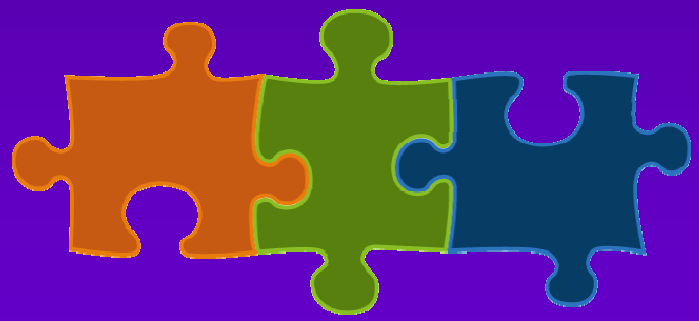
Users

- To discover projects
  - Read the project home pages
  - Download releases and milestones
  - Monitor the blogs (http://www.planeteclipse.org/)
  - Subscribe to the project newsgroup
  - Watch the screencasts
  - Try the tutorials
  - Use a distribution
- About the future
  - The Eclipse Roadmap (linked from home page)



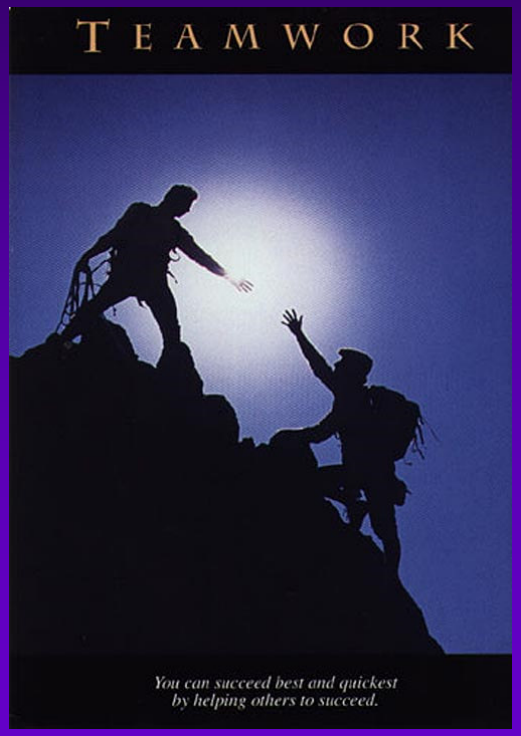
Plug-in Developers

- Your friends are:
  - Developer mailing lists
  - Project newsgroups
  - Weekly integration builds
  - API documentation
    - Your clear and detailed feedback on the APIs
  - Bugzilla
  - Patch files



Contributors

- Eclipse projects are meritocracies … translates to *you have to prove your abilities before being accepted*.
- Start with well-formed bug reports and feature requests.
- Build a good reputation with the existing committers on a project.
- Propose code enhancements and volunteer time to the project.
- After demonstrating your skills, ask a current project committer to sponsor you as a committer. The election process is defined in the project's charter.



Members

- If you'd like to become a Member of the Eclipse Foundation, start with
  http://www.eclipse.org/org/how-to-join.html
- Explains the necessary qualifications
  - Must release a Eclipse-based offering within 12 months
- Explains the benefits
  - Varies by level; Strategic members have Board seats, etc.
- Link to the application form and an email address for more information
  membership@eclipse.org

Questions?

www.eclipse.org/org/processes/