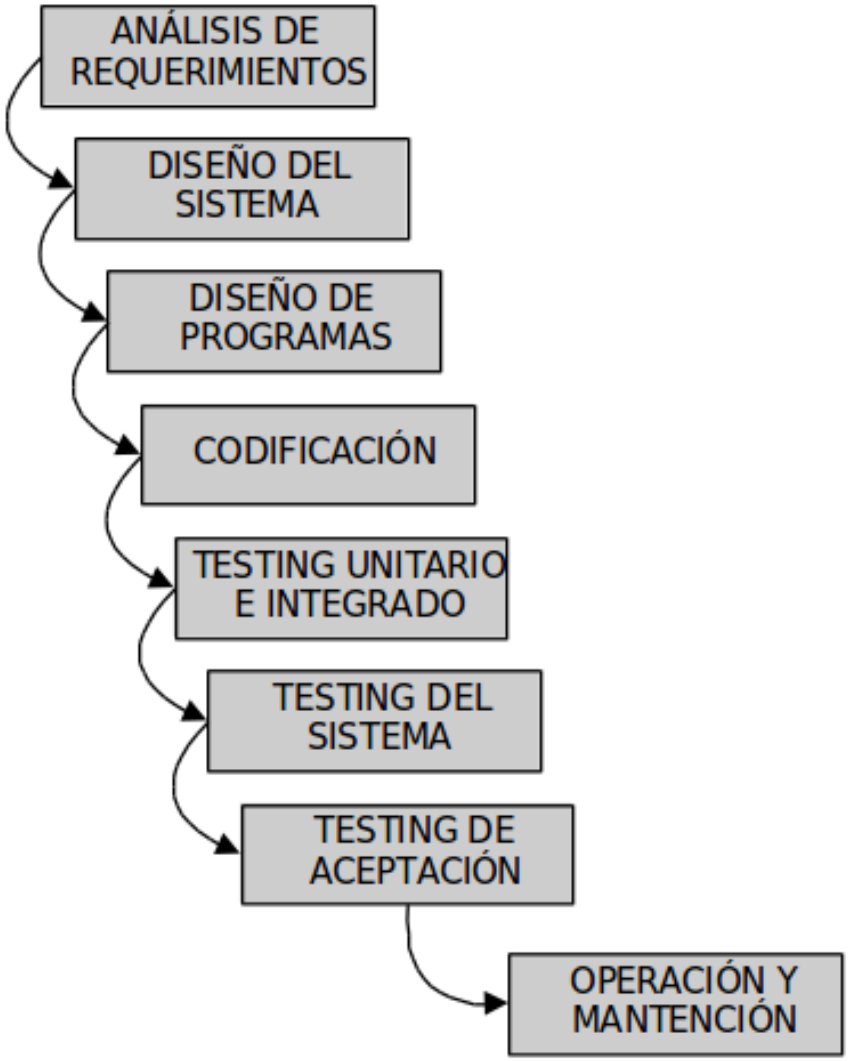
Ingeniería de Software I
Mantención y Evolución

Jocelyn Simmonds

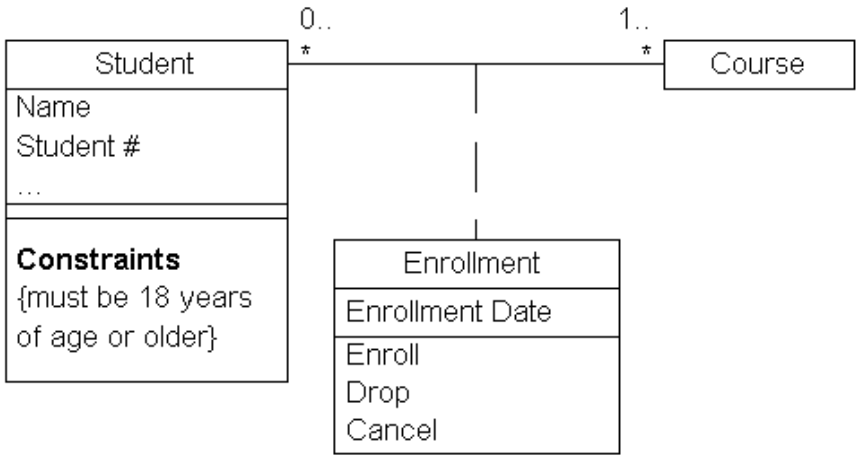
Departamento de Ciencias de la Computación

Fases de desarrollo

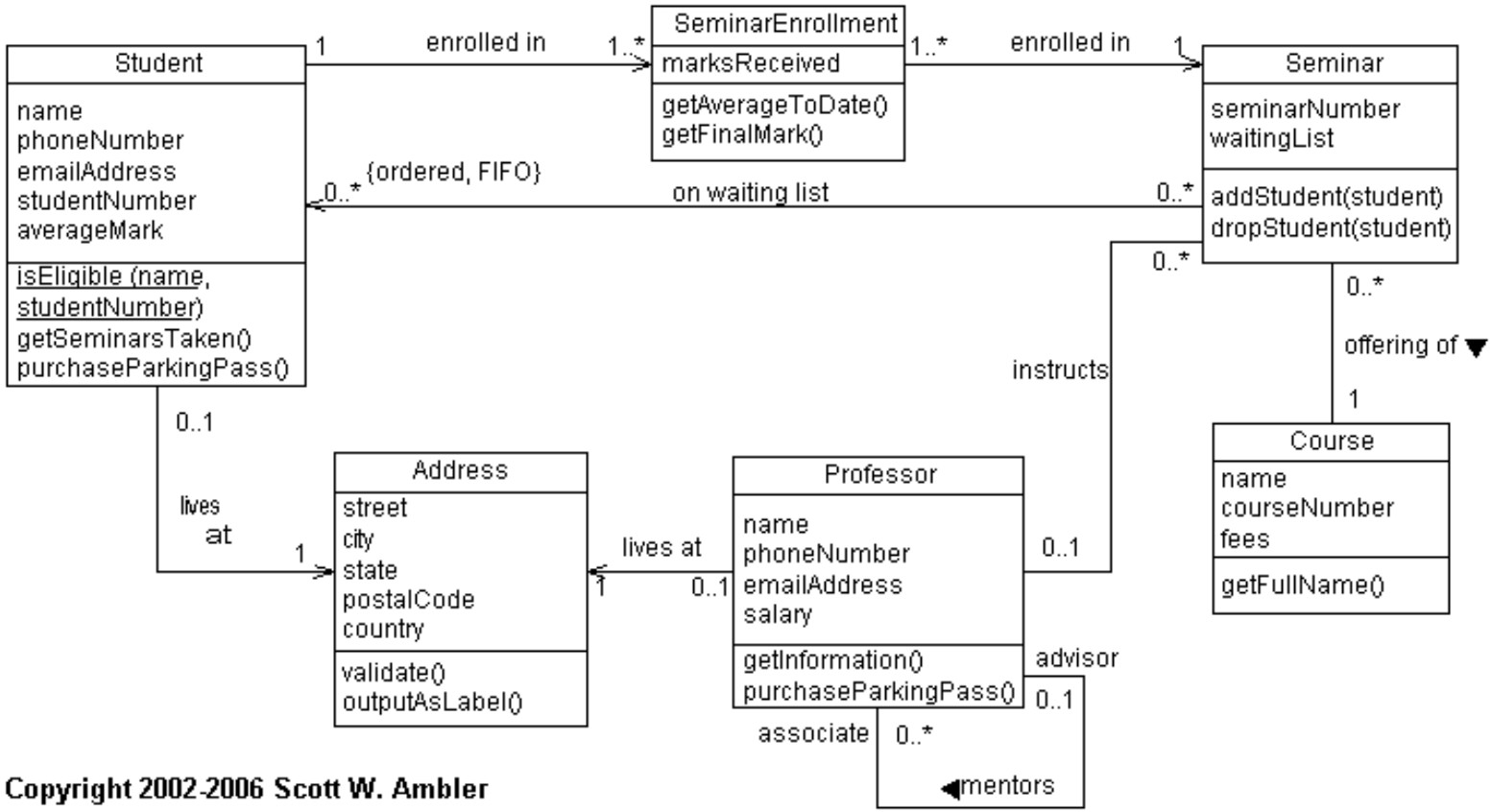


Ya hemos hablado un poco de requisitos, diseño e implementación . . . pero antes de seguir con testing, primero hablemos de mantención.

Empezamos con el "mejor" diseño para un producto...



Una vez que comienza la implementación . . .



Vemos la necesidad de agregar, cambiar o quitar elementos del diseño.

En realidad ...

Eventualmente van a estar a cargo de definir los requisitos o diseño, lo más probable es que empiecen corriegiendo bugs.

Lehman: leyes acerca de la evolución de SW (1/3)

Definidas entre 1974 y 1996:

1. Continuing Change: un sistema debe ser adaptado en forma continua, o sus usuarios se van a volver progresivamente menos satisfechos con el sistema
2. Increasing Complexity: a medida que un sistema evoluciona, su complejidad aumenta a menos que se trabaje para mantener o reducir esta complejidad
3. Self Regulation: el crecimiento de un sistema durante un proceso de evolución auto-regula, tendiendo a una distribución normal de artefactos (crecimiento lento al principio y fin, comparado con lo que ocurre entremedio)

Lehman: leyes acerca de la evolución de SW (2/3)

4. Conservation of Organisational Stability (invariant work rate): la tasa de actividad global efectiva promedio en un sistema en evolución es invariante sobre la vida útil del producto
5. Conservation of Familiarity: a medida que un sistema evoluciona, todas las personas involucradas (desarrolladores, usuarios, etc.) deben mantenerse al tanto de las nuevas versiones. El crecimiento excesivo hace difícil esta tarea, por lo que el crecimiento es invariante
6. Continuing Growth: la funcionalidad que ofrece un sistema debe incrementarse en forma continua para mantener satisfechos a los usuarios

Lehman: leyes acerca de la evolución de SW (3/3)

7. Declining Quality: la calidad de un sistema disminuye en el tiempo, a menos que sea mantenido en forma rigurosa y sea adaptado frente a cambios en su ambiente
8. Feedback System: los procesos de evolución son "multi-level, multi-loop, multi-agent feedback systems", y deben ser tratados como tal para lograr cualquier mejora razonable
   - hay que hablar con clientes, usuarios, equipo, etc.
   - hay que colectar datos acerca cambios en los requisitos
   - y también acerca de las prioridades a futuro
   - ...

Code decay

Se refiere al lento deterioro del rendimiento o capacidad de respuesta de un sistema a lo largo del tiempo . . .

- . . . haciendo que el sistema se vuelva defectuoso o inutilizable
- un sistema en esas condiciones pasa a ser un sistema "legacy"
- y necesitamos una nueva versión o nuevo producto para reemplazarlo

Code decay también se conoce como:

- software rot, code rot, bit rot, software erosion, software decay o software entropy

El costo de mantención

Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Caso especial: Y2K (Kappelman, 1998)

Table 1. Cost of Y2K compliance, measured as a percentage of annual IS budget, project progress, and percentage of the manager's time by economic sector

Sector	n	Y2K Cost as % of Annual IS Budget	Annual IS Budget	Gross Revenue	Y2K Project Progress* (Rank)	Project Manager % Time (Rank)
Agriculture	1	75	4.0 M	5.06 B	2.0 (15)	10.0 (14)
Military	3	40	106.7 M	5.8 B	4.7 (5)	69.3 (1)
Other	16	37	17.1 M	2.08 B	3.9 (9)	25.5 (11)
Banking, Securities, Investment	4	33	60.7 M	327.00 M	5.8 (1)	48.0 (5)
Insurance	16	28	63.2 M	3.07 B	5.3 (2)	45.2 (6)
Utilities	9	27	26.0 M	1.35 B	4.6 (6)	43.7 (7)
Retail	2	25	8.0 M	10.40 B	5.0 (3)	57.5 (2)
Consumer Good Manufacturing	11	23	32.6 M	1.59 B	3.1 (13)	22.5 (12)
Printing, Publishing	4	22	13.9 M	676.00 M	4.5 (7)	26.5 (10)
Government	6	21	44.7 M	2.14 B	3.8 (10)	39.8 (8)
Health	8	21	25.0 M	2.18 B	3.6 (12)	32.5 (9)
Entertainment	1	13	15.0 M	3.00 B	1.0 (17)	0.0 (18)
Capital Goods Manufacturing	9	11	24.7 M	3.38 B	4.8 (4)	12.3 (13)
Transportation	3	11	24.8 M	432.00 M	3.7 (11)	55.0 (3)
Food Services	3	3	25.5 M	13.40 B	2.5 (14)	6.7 (15)
Legal	1	0	6.0 M	500.00 M	1.0 (17)	1.0 (17)
Education	1	0	2.0 M	130.00 M	2.0 (16)	0.0 (18)
Business Services	2	–	2.6 M	0.00	1.0 (17)	49.5 (4)
Construction	1	–	700,000	55,000	4.0 (8)	2.0 (16)
Printing, Publishing	4	22	13.9 M	676.00 M	4.5 (7)	26.5 (10)
TOTALS	102	26.0%	$33.5 M	$2.73 B	4.1	32.3
Overall Sample Totals	180	24.6%	$42.3 M	$5.80 B	4.3	31.0

Abbreviations: M = $millions B = $billions * = 1–9 Scale n = sample size

99 little bugs in the code.

99 little bugs.

Take one down, patch it around.

127 little bugs in the code...

Más números

- Corbi (1989): el 60% del tiempo se usa para entender código existente
- Brooks (1975): el 20 - 50% de los bug fixes introducen nuevos bugs
  - Purushotaman y Perry (2005) dicen que el el 40% ...

Concepto de "maintenance programmer": un programador cuyo prinicpal labor es modificar el código que alguien más ha escrito

Tipos de mantención

- Perfective: introduce nuevas funcionalidades en el sistema
- Adaptive: adapta el sistema a un nuevo ambiente (nuevo sistema operativo, etc.)
- Corrective: corrige defectos de software
- Preventive: pensando en posibles cambios futuros (refactoring, etc.)

Más de la mitad de los cambios son perfectivos. La introducción de nueva funcionalidad también se conoce como "evolución" de software.

Tipos de cambios

Considerando el impacto que tienen sobre la funcionalidad:

- Cambios incrementales: agrega nueva funcionalidad, aumentando el valor del sistema
- "Podado" del código: quita funcionalidad obsoleta. No aumenta el valor del sistema, pero baja el "bloat" de este
- Reemplazos: se reemplaza una funcionalidad por una nueva (corrección de errores). Usualmente es un cambio incremental
- Refactoring: no hay cambios en la funcionalidad, sino que se hacen mejoras estructurales. No aumenta el valor del sistema, pero aumenta su mantenibilidad.

Tipos de cambios

Considerando el impacto que tienen sobre el sistema:

- Cambios localizados: solo afecta unos pocos módulos (pequeñas correcciones de errores, cambios anticipados en el diseño)
- Cambios no localizados: afecta 12 o más módulos (se agrega una cantidad significativa de nueva funcionalidad)
- Cambios masivos descentralizados: afecta gran parte del sistema (cambio de diseño fundamental, cambios en el sistema operativo, Y2K, etc.)

Estrategias de cambio

Quick fix:

- Los arreglos a la rápida usualmente significan alguna degradación en la estructura del sistema, y tienen costos a largo plazo
- Ejemplo típico: copy-paste de un método con algunas pequeñas modificaciones, en vez de modificar el método existente para manejar un caso más general $\rightarrow$ nos llenamos de metodos similares pero distintos $\rightarrow$ baja la entendibilidad del código

Los cambios deben estar bien diseñados, deberían mejorar la estructura del software y no degradarla.

Proceso de cambio de software

Necesitamos un proceso de cambio de software . . . y este proceso debe funcionar aun cuando la calidad del código no es la mejor

Proceso de cambio de software

- Initiation: llega un change request, o se selecciona un nuevo requisito del backlog
- Concept Location: el programador debe buscar el módulo donde debe implementar el cambio
- Impact Analysis: determinar cuales otros módulos son afectados por el cambio

Proceso de cambio de software

- Prefactoring: cambios estructurales y no de función, para facilitar la actualización
- Actualization: actualizar módulos existentes, agregar nuevos módulos, propagar los cambios a los módulos identificados en Impact Analysis
- Postfactoring: cambios estructurales y no de función, para reducir code decay

Proceso de cambio de software

- Verification: determinar que la nueva funcionalidad esta correctamente implementada, y que el código anterior aun funciona
- Conclusion: add $\rightarrow$ commit $\rightarrow$ push, y actualizan la documentación, etc.

¿Qué es un change request?

Un change request es un cambio que debe hacerse al software:

- cambios en los requerimientos
- Bug reports
- Refactorings preventivos
- etc.

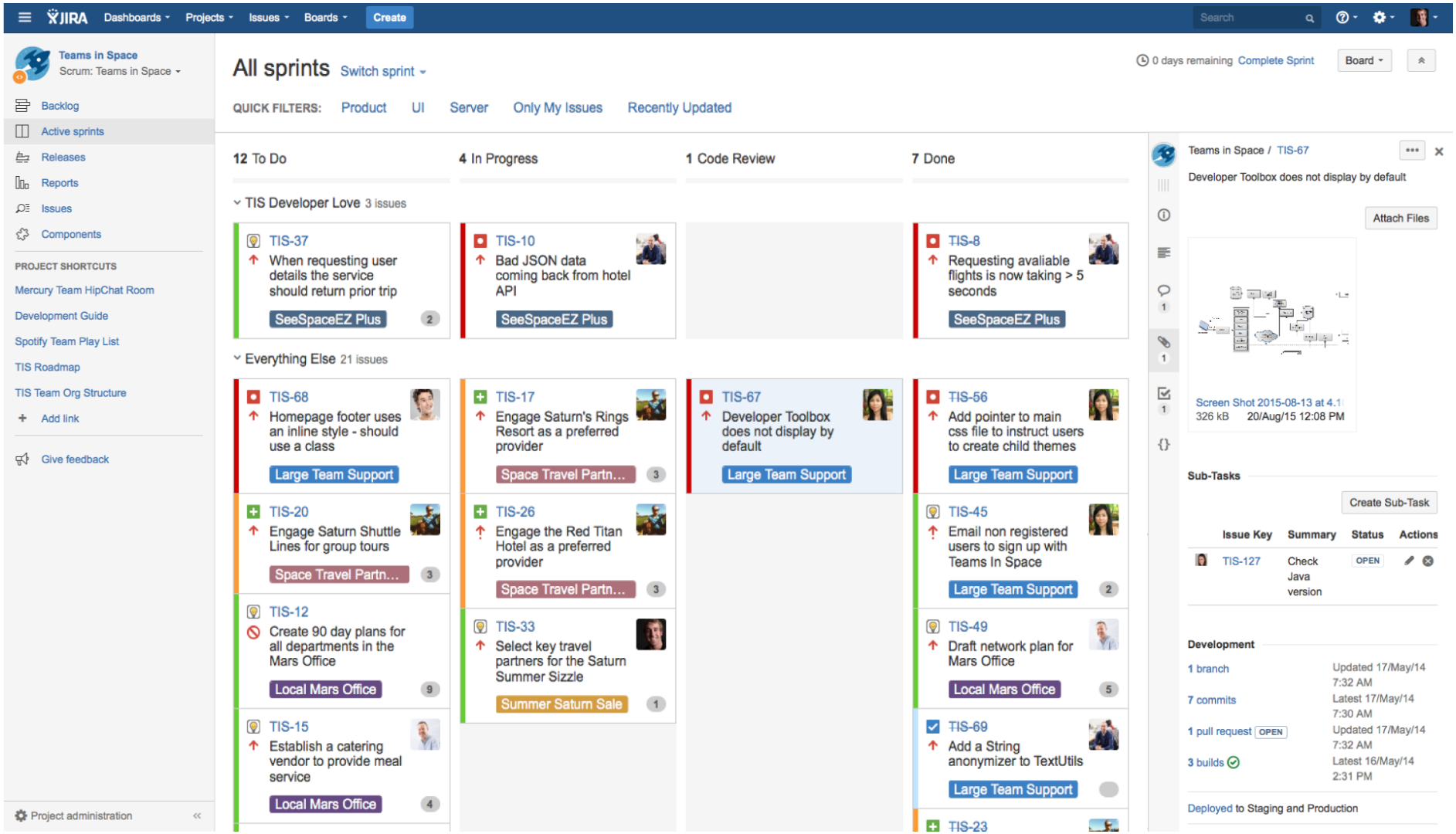
Especificados en lenguaje natural, casos de uso, historias de usuario, etc.

¿Cómo elegimos los change request?

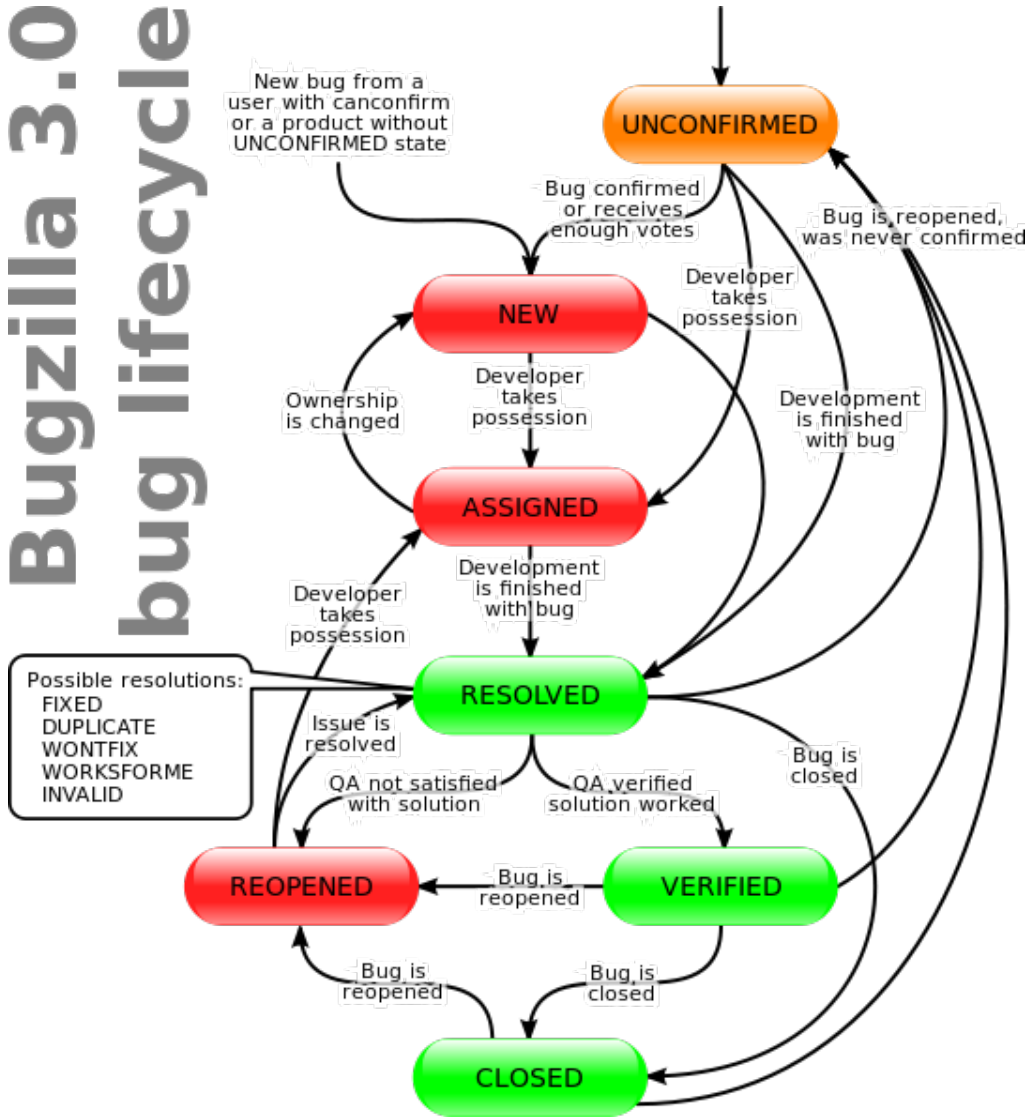
Ordenar por prioridad:

- En el caso de funcionalidad nueva: por el valor que aportan al sistema
- Ordenar bugs por gravedad:
  1. errores fatales: la aplicación no puede seguir funcionando
  2. bug serio: la tarea no se puede realizar en forma alternativa
  3. bug: la tarea se puede realizar en forma alternativa
  4. minor issue: no afecta la funcionalidad principal del sistema
- Es urgente corregir los errores fatales y los bugs serios
- También hay que tomar en cuenta el costo/beneficio de cada cambio
  - ¿vale la pena corregir un cierto minor issue? o estamos trabajando ya en una versión nueva donde este issue ya no será un problema?

Apóyense de alguna herramienta para este proceso



Estado de un bug en Bugzilla



Para resumir ...

- En general, los programadores deben trabajar con código que ellos no han escrito
- Con el tiempo, hasta el mejor diseño empieza a decaer en calidad
- La refactorización ayuda a mejorar esta situación
- La idea es de tratar de mantener la complejidad bajo control frente a cambios que deben realizarse